TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

MIR MIR ABDUL RASOOL KHAN

RECURRENT NEURAL NETWORKS FOR MODELING MOTION CAPTURE DATA

Master of Science thesis

Examiner: Heikki Huttunen, Atanas Gotchev
Examiner and topic approved by the Faculty Council of the Faculty of Computing and Electrical Engineering on 24th May 2017

ABSTRACT

MIR MIR ABDUL RASOOL KHAN: Recurrent Neural Networks for Modeling Motion Capture Data
Tampere University of Technology
Master of Science thesis, 54 pages, 1 Appendix page
May 2017
Master's Degree Programme in Information Technology
Major: Data Engineering
Examiners: Heikki Huttunen, Atanas Gotchev
Keywords: motion capture, recurrent neural network, LSTM, deep learning, generative model

This thesis introduces a Recurrent Neural Network (RNN) framework as a generative model for synthesizing human motion capture data. The skeleton model of the data used is a complex human skeleton model with 64 joints, resulting in a total of 192 degrees-of-freedom, which makes our data nearly three times as complex as in previous approaches of applying neural networks on motion capture data. The RNN model can generates good quality and novel human motion sequences that can, at times, be difficult to visually distinguish from real motion capture data, which demonstrates the ability of RNNs to analyze long and very high-dimensional sequences. Additionally, the synthesized motion sequences show strong inter-joint correspondence and extending up to 250 frames. The quality of the motion and its accuracy is analyzed quantitatively through various metrics that evaluate inter-joint relationships and temporal joint correspondence.

PREFACE

This thesis was done as a part of a Master's degree in Data Engineering at the Tampere University of Technology.

I would like to thank my advisor Heikki Huttunen for his guidance and supervision of this thesis and his support throughout the process. I am also grateful to him for introducing me to the 3D team which made this thesis possible, taking me one step closer to my long-term future goals in research in this area. I really appreciate that he helped me to find a thesis topic that is most related to my long-term future research plans.

I would also like to thank Atans Gotchev and Olli Suominen for their support and feedback. I would also like to thank them for accepting this thesis topic.

Mir A. Khan
May 24th, 2017

CONTENTS

LIST OF ABBREVIATIONS AND SYMBOLS

AI	Artificial Intelligence
ANN	Artificial Neural Network
BP	Back-Propagation
BPTT	Back-Propagation Through Time
MLP	Multi-Layered Perceptron
RNN	Recurrent Neural Network
FFN	Feed-Forward Network
FPS	Frames Per Second
GPU	Graphics Processing Unit
LSTM	Long Short-Term Memory
MSE	Mean Squared Error
MAE	Mean Absolute Error
RMS	Root Mean Squared
SGD	Stochastic Gradient Descent

NOTATION

$\mathbf{x}$	Vector
$\mathbf{X}$	Matrix
x	Scalar value
$\mathbf{x}^T$	Transpose of a vector $\mathbf{x}$
$diag(\mathbf{x})$	Function that converts a vector $\mathbf{x}$ to a diagonal matrix.

1. INTRODUCTION

Producing natural character animations is crucial for the success of many visual entertainment industries such as video games and film. In the case of film for example, it is important that the characters' movements appear natural and convincing to the viewers. Motion capture is one method of reproducing convincing animations by recording motion played by human actors. Motion capture data is used to animate 3D characters by mapping the same motion onto the virtual character. While motion capture offers many advantages, it has several drawbacks as will be discussed in this section. Alternative techniques for producing 3D animations include keyframing [29], simulation-based approaches [52][17][50], and in some cases, though impractical, it can be done manually.

1.1 Motion Capture

Motion Capture is the process of digitally recording 3D motion as played by an actor or an object. The recording process involves the use of hardware such as cameras, physical markers, and a special suit. Additionally, software that processes the data is used in conjunction with the hardware. Usually the motion capture data stores motion information in the format of joint angles with reference to a predefined skeleton with fixed bone lengths. Motion capture data is commonly used to animate 3D humanoid models for entertainment applications such as film and video games. This is an alternative to manually animating 3D characters, with the advantage of producing realistic animations and in most cases with less effort and faster results when compared to other techniques[20]. Several different motion capturing systems exist, but we will focus on optical systems because it is the medium in which our data set was recorded.

With passive optical motion capture systems, the subject wears a suit as shown in Figure 1.1 with markers placed on the subject's body, typically near the joints. The subject is surrounded with an array of cameras in a circular arrangement that detect these markers and then determine their positions using triangulation. These systems have the advantage of recording motion at high sampling-rates and allow-

Figure 1.1 *An actor wearing a special suit with markers used with the motion capture system. Photo by T-tus / CC BY*

ing the subject to move freely during the recording process, as opposed to other motion capture systems (non-optical) which restrict the movement of the actor. Optical motion capture systems have their drawbacks, however. The markers can be blocked by the subject which would prevent the camera from capturing it, resulting in recording inaccuracies [33][59]. Another problem is that optical-systems can be very expensive.

Difficulty of reusing the motion capture data poses another challenge. Processing the motion capture data for reuse in different scenarios can be difficult. For example, a walking animation in a straight line can easily be reused to represent walking from any two points on a level plane. This is trivially achieved by rotating the motion data around the Y axis and applying global translation to the desired new starting point. However, if we want to represent faster walking or running animations, reusing the same walking motion sequence would require laborious processing, and in some cases this is not possible without replacing all the content of the motion data by a completely different motion data. Additionally, it is not always possible to map the same motion data on different characters with different morphologies such as length of limbs and width of its body.

1.2 Previous and Related Work

We provide an overview of previous and related work related to methods for reusing motion capture and generating 3D human animations. We start by giving a review of kernel-based techniques for motion synthesis. Then we proceed to review simulation-based animation techniques using controllers. Finally, a brief overview of deep learning and its applications to animations is provided. We summarize the main contributions of this thesis at the end of this section.

Kernel-based methods Kernel-based techniques such as radial basis functions (RBF) have been shown to be useful for motion synthesis by blending several different motion sequences from a similar category. A technique has been introduced in [53] based on a combination of RBFs and low-order polynomials that can generate motion sequences by blending motions from a similar category and interpolating between them. The motions are categorized by emotional expressiveness and the behavior present in the motion. These techniques were later extended in [53] for solving inverse-kinematics (IK) problems and to generate smooth locomotion animations [47]. RBFs can be susceptible to overfitting to the data, but this can be overcame as shown in [42] with the use of Gaussian Processes (GP). Other techniques utilize Gaussian Process Latent Variable Models (GPLVM) to predict a single frame of motion from a given frame [68], and to reduce the dimensionality of the motion data to allow interactive control of the motion using high-level parameters [19].

Physics-based animation Alternative methods to motion capture and manual animation are simulation-based methods for generating natural motion. The principle in these methods is to develop control strategies for humanoids and imaginary creatures through the use of optimization methods such as genetic algorithms. The control strategies are then optimized with respect a criteria (often called a *fitness function*) such as total distance traveled. An early example of these methods was introduced in [61], where virtual creatures with random morphologies develop relatively optimal control strategies such as swimming, running, jumping, and crawling depending on their environment, their own physical characteristics, and the defined fitness function. Similar techniques have been used to produce more controlled behavior such as bipedal gait animations in a simulated environment [52]. Using biomechanical constraints have been shown to produce even more convincing and natural-looking animations for bipedal virtual creatures [17]. The main drawback of these techniques is that the generated motion is not often exactly what the animator may desire, and recent developments have tried to solve this by generating these animations from high-level parameters [50].

Deep learning Techniques based on neural networks with multiple layers are known as deep learning. Neural networks, which are crude models of the information-processing behavior of the biological brain, are extremely versatile machine learning tools with an ever-growing range of applications. Techniques have been previously used with good results for tasks such as image classification [32][6], face-recognition [34][46][35], audio classification [68][31][8], and speech recognition [67][68]. Majority of these tasks have previously been considered to be tasks that can only be performed exclusively by humans. Deep learning techniques for synthesizing 3D human motion has drawn attention recently. An approach has been proposed in [26][27] to learn a space of valid human poses (called the motion manifold) using a convolutional-autoencoder network architecture. The framework can be used for novel motion synthesis, motion interpolation, and error-correction. A method that uses conditional Restricted Boltzmann Machines (cRBM) have been used to synthesize human gait animations [64][65]. Recurrent Neural Networks are a variation of regular neural networks with feedback loops where the inputs of previous time steps determine the input at future and present time steps. This temporal attribute of RNNs makes them well-suited for analyzing time-series data. RNNs have bee used for automatic music composition and analysis [9][43], image synthesis [18], and handwriting generation [15]. RNN architecture has also been introduced in [11] for human motion labeling and generating novel motion sequences. Another RNN-based generative model is presented in [7] that can produce dancing motion sequences in some given style.

Contributions The work of this thesis is most closely related to [7] and [11], but we focus on more complex human skeleton models and performing quantitative analysis for measuring the qualitative attributes of the generated motion. Previous approaches in applying neural networks in the domain of motion capture have focused on human skeletal models with 21 to 25 joints [26][27][7]. This thesis investigates the application of neural networks on skeletal models with 64 joints, making the data set used in this thesis nearly 3 times as complex as in previous methods. The thesis aims to demonstrate that recurrent neural networks (a variant of neural networks) can handle sequential data of very high dimensionality. The data we are using are sequences of 192-dimensional features, with a temporal length extending up to 200 discrete time steps. The contributions of this thesis can be summarized as follows:

- A generative RNN-based frame-work for motion synthesis for a complex human skeleton model with 64 joints, including hand fingers.
- Data augmentation techniques for motion capture data for machine learning

applications.

- Analysis and comparison of three different RNN networks for synthesizing motion sequences and an analysis of the generated motion sequences by the network.

1.3 Outline

In this section, we outline the organization of this thesis and describe the contents of each chapter in the order they appear in.

Chapter 2 introduces the general problem of supervised learning with an example for linear regression. Subsequent sections introduce Artificial Neural Networks and their notable variants, including Recurrent Neural Networks and Long Short-Term Memory.

Chapter 3 introduces the data preparation approaches, data augmentation techniques, motion prediction procedure, model selection criteria and the implementation environment.

Chapter 4 presents the results obtained using the methods outlined in Chapter 3 and provides a detailed analysis and evaluation of the results.

We conclude with Chapter 5 discuss plans and goals for future work in this area.

2. MACHINE LEARNING

The term *Machine Learning* (ML) covers a wide range of techniques and methods for the automatic analysis of data. Common applications of machine learning methods include data classification, regression, and clustering. One important concept in machine learning is Artificial Neural Networks (ANNs) and its many variants, which have widely expanded the field and its range of applications over the years. ANNs generally fall under the supervised learning category, which differs from unsupervised learning and other approaches in that the general desired behavior of the model (in this case the neural network) is usually known; in some areas of application, however, it was demonstrated that ANNs can be designed using unsupervised learning approaches such as genetic algorithms [61][41] and reinforcement learning [62]. This section begins with the introduction of the general problem of supervised learning, illustrated with an example. The basic ANN model, its variants and training approaches are then introduced in subsequent sections.

2.1 Supervised Learning

The basic problem of supervised learning can be stated as a pattern classification task, where we are given a set of N patterns or samples $D = \{(\mathbf{x}_i, z_i), i = 1, \ldots, N\}$, where $\mathbf{x}_i$ is an m-dimensional vector $\mathbf{x} = \begin{bmatrix} x_1 & x_2 & \ldots & x_m \end{bmatrix}^T$. Associated with each $\mathbf{x}_i$ is a class identifier $z_i \in \{\omega_1, \omega_2, ..., \omega_C\}$, which determines $\mathbf{x}_i$ to belong to one of C classes. D is usually called the training set (or the design set), and each pair $(\mathbf{x}_i, z_i) \in D$ denotes an individual training sample. Each class ω represents a particular and distinct class category. The exact definition and representation of a class is usually dependent on application, with one common example being a binary classification task where we determine whether a data sample $\mathbf{x}_i$ as belonging to some category or not. For example, if $\mathbf{x}$ is a digital image, then our classification model may be given the task to determine whether the image $\mathbf{x}$ contains a human face , with $z = \omega_1 = 1$, or that it does not contain a face, $z = \omega_2 = 0$.

The pattern vector $\mathbf{x}$ may be a vector of features (either designed by hand or automatically extracted) that characterize some possibly more complex data sample.

Examples for features in the case of images may include regions in the images, such as the eyes in the case of images of human faces. In some applications, however, $\mathbf{x}$ may contain the whole representation of the image, i.e., all the pixels. Typically in these cases, feature extraction is performed automatically by the model. One example of a model where feature extraction is performed automatically is Convolutional Neural Networks (CNNs), which are a variation of the more general model of Artificial Neural Networks (ANNs) whose details are to be discussed in the next section.

Typically, the dataset D is split into three parts: the first part, called the training set, is used to adjust and design the model parameters. The second and third sets, called the validation and test sets, serve similar roles but with notable distinctions. The validation set is used as a part of the model-selection stage and to tune the parameters of the network in order to avoid a phenomenon called *overfitting*. The validation set and its role in mitigating overfitting will be made more clear later in section 2.1.2. The third partition on the data set, called the test set, is used to evaluate the performance of the selected model to generalize on previously unseen samples.

The goal of supervised learning is to build a predictive model which can accurately classify each pattern as belonging to the correct class. The training set is used to build such a model, with the expectation that the model's performance on the training set is an indicator of its future performance on previously unseen samples. One could think of the supervised-learning approach as a procedure for teaching a model to *learn by example* and to gain an understanding of the general structure of the data.

2.1.1 Example: Multiple Linear Regression

A simple and classic example for introducing supervised learning is linear regression. Suppose we are given the data set in table 2.1, showing data collected about scores on the Scholastic Assessment Test (SAT), Graduate Record Examination (GRE), and the final mathematics exam in some semester. We wish to design a predictive model that can accurately predict the score y for some previously unknown report of the SAT score x_1 and a GRE score of x_2. We use the following linear polynomial as our model model:

$$y = w_1 x_1 + w_2 x_2 + b \tag{2.1}$$

SAT	GRE	Final Exam
1	25	100
10	38	95
5	4	55
3	5	68
7	2	65
10	40	55
1	4	48
11	22	98
8	23	46
6	30	68
7	21	37
3	23	89

Table 2.1 *Data on exam scores as reported by the students.*

The parameters w_1, w_2 are often referred to as the *weights*. The parameter b is known as the *bias* term. By letting $w_0 = b$ and $x_0 = 1$, we can express equation 2.1 in the more compact form

$$y = w_0x_0 + w_1x_1 + w_2x_2 = \sum_{i=0}^{m} w_ix_i = \mathbf{w}^T\mathbf{x} \tag{2.2}$$

We wish to find parameters w_0, w_1, and w_2 such that the model is optimally fitted to the training set D, with the expectation that the model can then accurately predict for some unknown x_1 and x_2 drawn from a similar data pool. This can be done by minimizing some error function with respect to our model's parameters. We will choose the error energy as our error function which is given as

$$\mathcal{E}(w) = \frac{1}{2}\sum_{n=0}^{N}(y_n - d_n)^2, \tag{2.3}$$

where d_n is the desired output value for the model for training sample n. Note that this is a convex optimization problem whose closed-form solution can be easily solved for. However, an iterative procedure is more desirable in order to generalize to other non-convex problems (typically the case in deep neural networks) for which a closed-form solution is intractable. Let us use the Least Mean Squared (LMS) algorithm for the purpose of this example, which uses gradient descent for optimization on the randomly initialized parameters. We will illustrate this process on one particular training example $\mathbf{x}_i$ as follows

$$\frac{\partial \mathcal{E}(w)}{\partial \mathbf{w}} = \begin{bmatrix} \frac{\partial \mathcal{E}(w)}{\partial w_0} & \frac{\partial \mathcal{E}(w)}{\partial w_1} & \frac{\partial \mathcal{E}(w)}{\partial w_2} \end{bmatrix}^T \tag{2.4}$$

$$= \begin{bmatrix} (y_i - d_i)x_0 & (y_i - d_i)x_1 & (y_i - d_i)x_2 \end{bmatrix}^T \tag{2.5}$$

$$= (y_i - d_i)\mathbf{x}_i \tag{2.6}$$

Then, this leads to the following update rule for adjusting the weights for this model

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha(y_i - d_i)\mathbf{x}_i \tag{2.7}$$

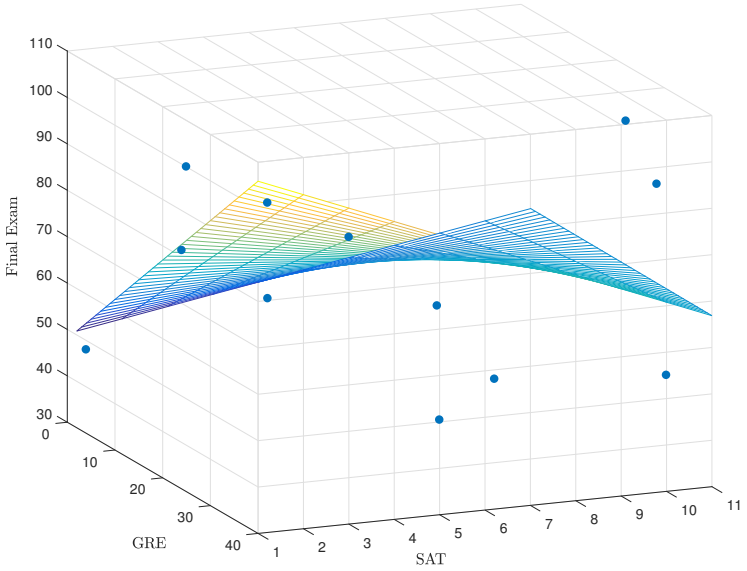

Figure 2.1 *Fitted model*

The parameter α is a positive constant that controls the magnitude of updating the parameter using this rule. Figure 2.1 shows the fitted model to the data set in table 2.1. For this specific problem, performing these updates iteratively is guaranteed to converge (as long as the learning rate is small enough) since there is only one global minimum.

2.1.2 Overfitting

A phenomenon known as overfitting arises in supervised learning problems when the model is accurately fitted to the training data samples, but it fails to generalize on previously unseen samples. Several techniques and good practices can be used to identify and mitigate the problem of overfitting. One common approach is the use of a cross-validation set (or simply validation set). A validation set can help reveal if a model has overfitted the training data by adding an additional testing phase to the model-design procedure. We first determine a set of models with a different set of parameters for each of them. Each model is then trained on the same training set. Once training is completed, the performance of each trained model on the validation set is evaluated. Essentially, the validation stage serves to provide a comparison of the performance of the model as its complexity changes. The model that performs best on the validation set is then selected to be tested on the final test set, which will determine how well the model is able to generalize on unseen examples. The performance on the validation set cannot be used as a performance measure of the model since the validation set was already used in the model selection stage. Another use for the validation set is to determining when to stop training the model. For example , we may test our model on the validation set after each training step. Once the error on the validation set stops decreasing, then we can stop training the model.

Regularization One way to avoid overfitting is using regularization, which is a technique that adds a penalty term in the loss function of a model. This term penalizes proportionally to the complexity of the model, therefore favoring models with smaller parameters. Regularization can be performed by modifying our loss function as follows

$$\mathcal{E} = \frac{1}{2}\sum_{i=1}^{N}(d_i - y_i) + \lambda \sum_{j=1}^{n} w_j^2 \tag{2.8}$$

Here, N is the number of training samples in our problem, and n is the number of parameters of the model excluding the bias. The parameter λ is a constant that controls the contribution of the penalty term to the loss function. The larger we make λ the more of an impact it has on the error and therefore how the weights are updated during training.

Data Augmentation When our data is scarce, we can use data augmentation which is the technique of applying domain-preserving transformations or modifica-

tions on data with the condition that the data retains its label. The purpose of data augmentation is to expanding the size of the training data by generating new data samples from the existing data set, which can help mitigate the problem of over-fitting. These techniques in image classification applications have been shown to improve the ability of a model to generalize [6] [60]. In the case of images, this might include applying spatial transformations to the data such as rotation, flipping, and in cases of large images, cropping. Figure 2.2 shows an example for data augmentation in image-classification applications and it illustrates the label-invariance of the sample "X" to 3 different spatial transformations, while the label is only preserved for one of the transformations for the sample "Y". For the sample corresponding to the letter "Z", none of the transformations retain its label and, in fact, the 3rd transformation results in a sample appearing like the letter "N".

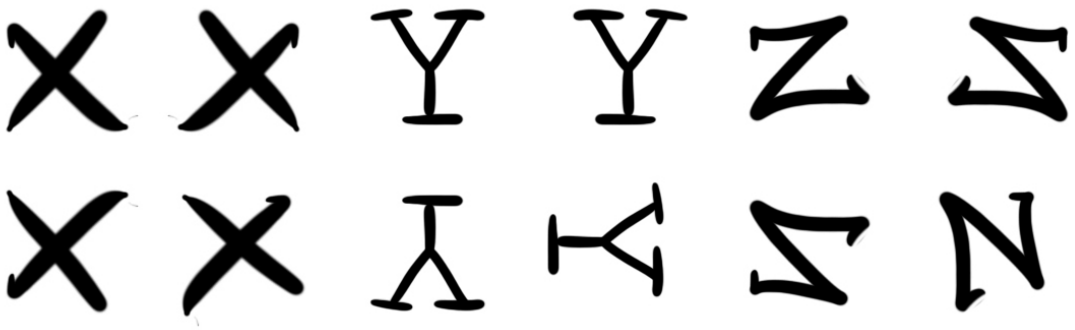

Figure 2.2 *Sample images corresponding to the labels "X", "Y" and "Z" from a theoretical data set of hand-written English characters. Each sample is transformed through horizontal flipping, vertical flipping, and a* 90*-degrees clock-wise rotation.*

2.2 Perceptron

The first model of a neuron as a computational unit was proposed by McCulloch Pitts in 1943 [38]. The operating principle of this neuron was basically to compute a weighted sum of input signals and compare it to a threshold. This is illustrated in Figue 2.3 A total sum value that is larger than the threshold would result in the neuron outputing a value of 1, and a value of 0 otherwise. The problem with this model was the lack of a learning algorithm that can determine to parameters for the neuron in order to perform its tasks. This problem was solved in 1958 by Frank Rosenblatt [54] by introducing a learning rule that can automatically determine the appropriate parameters.

The first component of the perceptron is a set of synapses or connecting links, each with an associate weight value. The neuron computes its induced local field v as a linear combination between its weights w and the input signal x

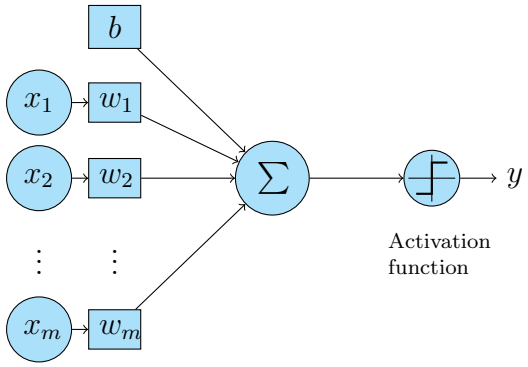

Figure 2.3 *Neuron model.*

$$v = \sum_{i=1}^{m} w_i x_i + b \tag{2.9}$$

Here b is a free parameter called the bias. This expression can be simplified by letting $x_0 = 1$, so that we can rewrite v as:

$$v = \sum_{i=0}^{m} w_i x_i = v = \mathbf{w}^T \mathbf{x} \tag{2.10}$$

The second element in the perceptron is an activation function ϕ, which computes the output of the perceptron $y = \phi(v)$. The activation function may be chosen depending on the desired behavior of the perceptron. A list of commonly used activation functions with a brief explanation is provided below.

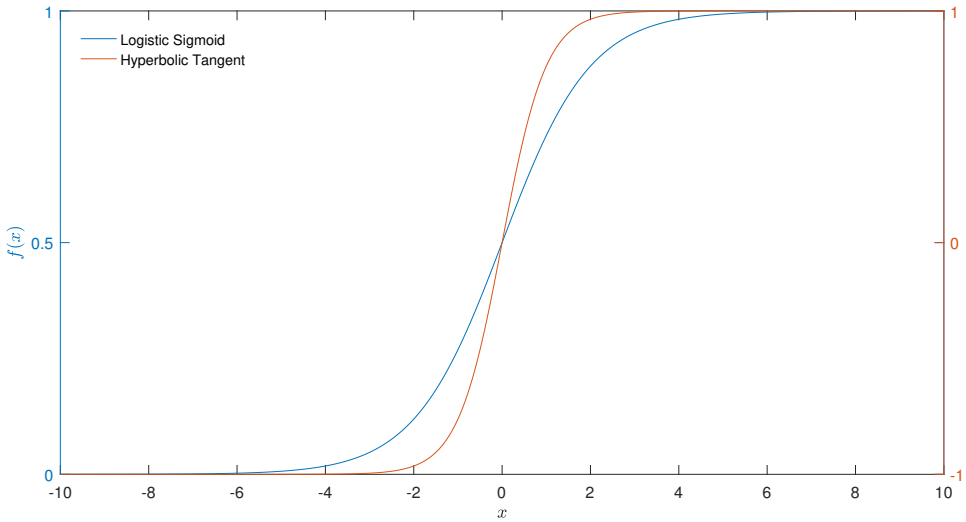

Figure 2.4 *Plots of the logistic sigmoid and hyperbolic tangent functions.*

- The step function is a discrete-valued function that outputs either a 0 or a 1.

$$s(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \end{cases} \tag{2.11}$$

- The logistic-sigmoid is a continuous function that compresses the value of its input between the range 0 and 1. It can be thought of as a smooth step function, and it is expressed as

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.12}$$

- The tanh(x) function (also known as the hyperbolic tangent) is a bipolar version of the logistic sigmoid function. The graph of the hyperbolic tangent and the logistic sigmoid are shown in Figure 2.4. The hyperbolic tangent function is given by the expression

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{2.13}$$

- The ReLU [14] activation function offers several advantages, among these are the efficient propagation of the gradient which avoids a phenomenon known as the vanishing gradient in deep networks which will be discussed in Section 2.4. The ReLU(x) function is expressed as

$$\text{ReLU}(x) = \max(0, x) \tag{2.14}$$

- Softmax can output posterior probabilities. Usually used at the output of clustering system. It can also be used at the output of a classification system to provide certainty or confidence values. The softmax function is given by the expression

$$\text{softmax}(x_j) = \frac{e^{x_j}}{\sum_m e^{x_m}} \tag{2.15}$$

2.3 Feed-forward neural networks

A multi-layered perceptron (MLP) or an Artificial Neural Network (ANN) is made of many such neuron units which can make them substantially more expressive and powerful. Essentially, an MLP is an information processing system, comprised of fundamental units called the "perceptron" (also called "neuron"), which is a simplified model of a biological neuron to serve as an individual information-processing

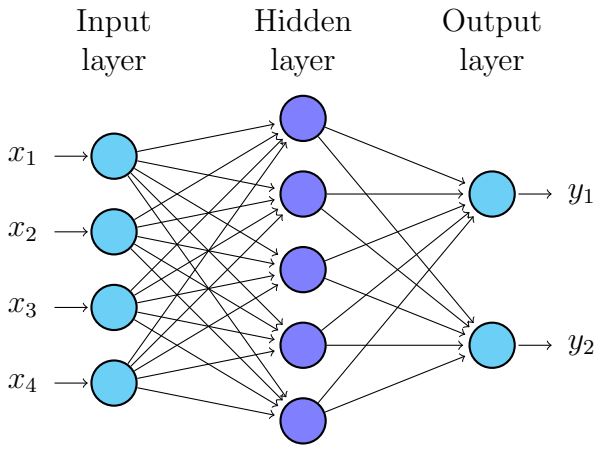

Figure 2.5 *Feed-Forward Network with one hidden layer, not including bias terms.*

unit. MLPs have found great utility in a wide range of application domains and they have been previously shown to achieve good results in tasks such as image classification [32][6], face-recognition [34][46][35], audio classification [68][31][8], and speech recognition [67][68].

Each layer in an MLP computes a new representation of the input, ending with the output layer. The simplest MLP has one hidden layer, but it can have many such layers with a variable number of neurons in each. The most commonly used MLP architecture is the Feed-Forward Network (FNN) architecture, where the input signal flows from the input layer to the output layer. A diagram of such a network is shown in Figure 2.3. The operation of the network is as follows: the network receives an input signal $\mathbf{x}$ from the input layer, feeds the input through each layer and computes an output signal. The output of the final layer $\mathbf{y}$ is the output of the network.

Figure 2.5 shows an MLP accepting inputs $\mathbf{x} \in \mathbb{R}^4$ and producing outputs $\mathbf{y} \in \mathbb{R}^2$. Essentially, the network can be seen as a function providing a non-linear mapping $f : \mathbb{R}^m \to \mathbb{R}^n$, where m and n are the dimensionality of the input and output signals respectively.

In order to illustrate the computation process in an MLP, the induced local field for some neuron of interest is restated as the dot product

$$v = \mathbf{w}^T\mathbf{x}, \tag{2.16}$$

where $\mathbf{w} = \begin{bmatrix} w_0 & w_1 & w_2 & \dots & w_m \end{bmatrix}^T$, and $\mathbf{x} = \begin{bmatrix} x_0 & x_1 & x_2 & \dots & x_m \end{bmatrix}^T$. The variable w_0 is the bias term for the neuron, and $x_0 = 1$. A layer in an MLP is completely

characterized by its weights matrix

$$\mathbf{W} = \begin{bmatrix} w_{0,0} & w_{0,1} & w_{0,2} & \dots & w_{0,n} \\ w_{1,0} & w_{1,1} & w_{0,2} & \dots & w_{1,n} \\ w_{2,0} & w_{2,1} & w_{0,2} & \dots & w_{2,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{m,0} & w_{m,1} & w_{m,2} & \dots & w_{m,n} \\ , & & & & \end{bmatrix} \tag{2.17}$$

where n denotes the number of hidden units in the layer characterized by this weights matrix. In the case of the network in Figure 2.5, $n = 5$. In order to express the output of the hidden layer of this network, we begin by first expressing the induced local field for all neurons in the hidden layer as the result of the matrix product

$$\mathbf{W}^T\mathbf{x} = \begin{bmatrix} w_{0,0} & w_{1,0} & w_{2,0} & w_{3,0} & w_{4,0} \\ w_{0,1} & w_{1,1} & w_{2,1} & w_{3,1} & w_{4,0} \\ w_{0,2} & w_{1,2} & w_{2,2} & w_{3,2} & w_{4,2} \\ w_{0,3} & w_{1,3} & w_{2,3} & w_{3,3} & w_{4,3} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \tag{2.18}$$

Then, if we let ϕ_h denote the activation function of the neurons in the hidden layer, then the output of the hidden layer is expressed as

$$\mathbf{h} = \phi_h(\mathbf{W}^T\mathbf{x}) \tag{2.19}$$

Then similarly, if U denotes the weights matrix of the output layer, the induced local field of all neurons in the output layer is expressed as the matrix product

$$\mathbf{U}^T\mathbf{h} = \begin{bmatrix} u_{0,0} & u_{1,0} & u_{2,0} & u_{3,0} \\ u_{0,1} & u_{1,1} & u_{2,1} & u_{3,1} \end{bmatrix} \begin{bmatrix} h_0 \\ h_1 \\ h_2 \\ h_3 \end{bmatrix} \tag{2.20}$$

Then the final output of the network can be computed as follows

$$\mathbf{y} = \phi_o(\mathbf{U}^T\mathbf{h}) = \phi_o(\mathbf{U}^T\phi_h(\mathbf{W}^T\mathbf{x})) \tag{2.21}$$

Typically, we compute outputs of *batches* of input signals, i.e. our input signal is a matrix of N input samples

$$\mathbf{X} = \begin{bmatrix} x_{0,0} & x_{0,1} & x_{0,2} & \dots & x_{0,N} \\ x_{1,0} & x_{1,1} & x_{0,2} & \dots & x_{1,N} \\ x_{2,0} & x_{2,1} & x_{0,2} & \dots & x_{2,N} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{m,0} & x_{m,1} & x_{0,2} & \dots & x_{m,N} \end{bmatrix} \tag{2.22}$$

Then the output matrix $\mathbf{Y}$ of any feed-forward MLP with $L-1$ hidden layers, where the Lth layer represents the output layer, can be expressed as the series of matrix products and function compositions

$$\mathbf{Y} = \phi_L(\mathbf{W}_L^T \dots \phi_2(\mathbf{W}_2^T\phi_1(\mathbf{W}_1^T\mathbf{X})) \dots) \tag{2.23}$$

In order for an MLP to be useful and perform a task, we have to determine its parameters that can allow it to do so. The next section introduces the most commonly used algorithm for designing the parameters of a neural network.

2.3.1 Back-Propagation Algorithm

The process of designing a predictive model (a neural network in particular) is typically referred to as *training*—an iterative procedure where the parameters of the network are adjusted after each training step (called *epoch*) until some stopping criteria is met. While several methods exist for training such as reinforcement learning and genetic algorithm, the most commonly used method is known as the Back-Propagation algorithm (BP). In order to gain a clear understanding of the algorithm, we will illustrate this process for a single neuron and for one particular weight. The algorithm is introduced in a manner similar to [22] which serves as an excellent reference on this topic and to which we refer readers interested in further details.

The original Back-propagation algorithm is an on-line learning algorithm that applies iterative and small increments in the parameters of the network (also called

weights) in each epoch, in such a manner that the network's performance on the most recently seen input sample is improved, with the expectation that this improvement will generalize over the larger data-set. The network's parameters are initialized to random values. In the simplest case, the operation of the algorithm may be summarized into two phases (more physically described as *passes*): first, the network computes an output from an individual sample, computes an error signal as the difference between the desired response and the actual response, and then adjusts the weights such that it produces outputs identical or closer to the desired response when shown similar or identical input samples in the future. The algorithm does not make any changes in the parameters during the forward pass. This error signal then propagates backwards through the network and the required change in the parameters is computed for each parameter. We will illustrate the operation of the BP algorithm on some particular neuron at the output layer of a network, and some neuron j at the hidden layer of the network.

BP performs weight updates using the Stochastic Gradient Descent algorithm, and its general formula is given as follows

$$w_{ki} \leftarrow w_{ki} + \Delta w_{ki} \tag{2.24}$$

Where w_{ki} denotes the value of the synaptic link connecting neuron i to neuron k, and

$$\Delta w_{ki} = -\eta \frac{\partial \mathcal{E}(w)}{\partial w_{ki}}, \tag{2.25}$$

where $\mathcal{E}(w)$ is the loss function. The parameter η, usually a positive constant scalar value, is called the learning rate which controls the magnitude of moving in the direction of the gradient. The expression in equation 2.25 points in a direction in the weight space that minimizes $\mathcal{E}(w)$. Note that this is very similar to the LMS algorithm which was discussed in section 2.1.1. The BP algorithm does this by taking a small step controlled in magnitude by η, and in a direction determined by gradient. For concreteness and convenience, we will restate some of the fundamental equations for this specific problem. Additionally, the functional dependence of $\mathcal{E}$ on w will be omitted for brevity.

The induced local field v for neuron k is defined as

$$v_k = \sum_{j=0}^{m} w_{kj} y_j \tag{2.26}$$

Where $m+1$ is the total number of input signals applied to neuron k including the bias. The output signal for neuron k at time step n is then given by

$$y_k = \phi_k(v_k) \tag{2.27}$$

If we denote by d_k the desired response of neuron k, then the error-signal for this neuron is defined by

$$e_k = d_k - y_k \tag{2.28}$$

As before, we will use the error energy as our loss function for mathematical convenience. For our problem, this would result in some slight modification to the expression

$$\mathcal{E} = \frac{1}{2} \sum_{j \in O} e_j^2 = \frac{1}{2} \sum_{j \in O} (d_j - y_j)^2, \tag{2.29}$$

where O denotes the set of all neurons at the output layer of the network. Using the chain-rule from calculus, we can express the gradient of this loss function as

$$\frac{\partial \mathcal{E}}{\partial w_{ki}} = \frac{\partial \mathcal{E}}{\partial e_k} \frac{\partial e_k}{\partial y_k} \frac{\partial y_k}{\partial v_k} \frac{\partial v_k}{\partial w_{ki}} \tag{2.30}$$

By letting

$$\delta_k = -\frac{\partial \mathcal{E}}{\partial v_k} = -\frac{\partial \mathcal{E}}{\partial e_k} \frac{\partial e_k}{\partial y_k} \frac{\partial y_k}{\partial v_k}, \tag{2.31}$$

and differentiation equation 2.26 with respect to w we obtain the expression

$$\frac{\partial \mathcal{E}}{\partial w_{ki}} = \delta_k y_k, \tag{2.32}$$

where the first term δ_k denotes the *local gradient* for neuron k and it is defined as

Accordingly, we may restate the delta update rule for the synaptic weight linking neuron i to k in the general and more expressive form

$$\Delta w_{ki} = \eta \delta_k y_k \tag{2.33}$$

For a neuron k at the output layer of the network, its local gradient is simply determined by differentiating equations 2.26- 2.28 , which results in

$$\frac{\partial \mathcal{E}(n)}{\partial w_{ki}(n)} = -e_k(n)\phi_j'(v_k(n))y_i(n) \tag{2.34}$$

Finally, for a neuron h at the hidden layer of the network, its local gradient δ_h can be shown to take the form

$$\delta_h = \phi_h'(v_h) \sum_{j \in H} \delta_j w_{hj}, \tag{2.35}$$

where w_{hj} is the weight on the synaptic connection connecting neuron j to neuron h, and H is the set of all hidden neurons in the preceding layer.

Mini-batch technique Standard SGD as an optimizer can be slow to converge. Several techniques have been proposed to remedy this problem. One alternative is to use batch gradient descent, where we update the parameters of the model after showing all the samples in our training set to the network, but this can be difficult when we have a large data-set and in many cases impractical due to computing resources limitations. The solution is to use mini-batches, i.e., using small sets from the training set and make updates to the parameters of the network after each mini-batch. This is also computationally efficient because the weights are updated less often and the computation of the gradient for each sample in the batch can be done in parallel on a GPU. The mini-batch approach is a good choice when he have a highly redundant and large data set.

Learning rate and momentum The momentum method [51] is a technique for accelerating the convergence of gradient descent where we compute the current gradient and then take a big jump in the direction of the updated and accumulated gradient. The idea behind the technique is to accumulate velocities in directions with gentle yet consistent gradients. The momentum technique can be implemented

by performing the following modification on the weight-updating rule

$$\Delta w_i \leftarrow \alpha \Delta w_i - \eta \frac{\partial \mathcal{E}}{\partial w_i}, \tag{2.36}$$

for some weight w_i. The parameter α is a positive constant called the *momentum constant*, usually set to 0.9.

RMSprop In neural networks with multiple layers, the learning rates for each parameter can vary widely. One technique that keeps this into account for improving the learning rate is RMSprop [66], which keeps a running-average of the squares (magnitudes) of recent gradients and then normalizes the current gradient by the root of this average. The new update rule for some parameter w_i is given as follows

$$\Delta w_i \leftarrow -\frac{\eta}{\sqrt{s}} \frac{\partial \mathcal{E}}{\partial w_i}, \tag{2.37}$$

where the running average s at the same time step is given by

$$s \leftarrow \alpha \Delta w_i - (\alpha - 1)(\frac{\partial \mathcal{E}}{\partial w_i})^2 \tag{2.38}$$

Here, α is a positive constant called the *momentum constant* and is usually set to the value 0.9.

2.4 Recurrent Neural Networks

Recurrent Neural Networks are MLPs with the additional component of feedback loops, where the output of the network at a particular time step n is dependent both on the input at time n and all previous inputs. This dependence on past inputs is represented implicitly through a *hidden state* that is computed at each time step. The temporal nature of RNNs makes them more biologically realistic and can allow them to remember information for a long time, which makes them well-suited for sequence analysis, and they have, in fact, been used as generative models in various domains [2][5][18][63]. RNNs can be much more expressive than standard MLPs and they have been shown to be equivalent to Turing machines in terms of computability [58][21], which leads to the profound implication that RNNs are theoretically capable of simulating any computer algorithm. Several variations of RNNs exist, among these are Elman networks [10], time delay neural networks [67], and Long Short-Term Memory [24], which we will introduce in more detail

shortly.

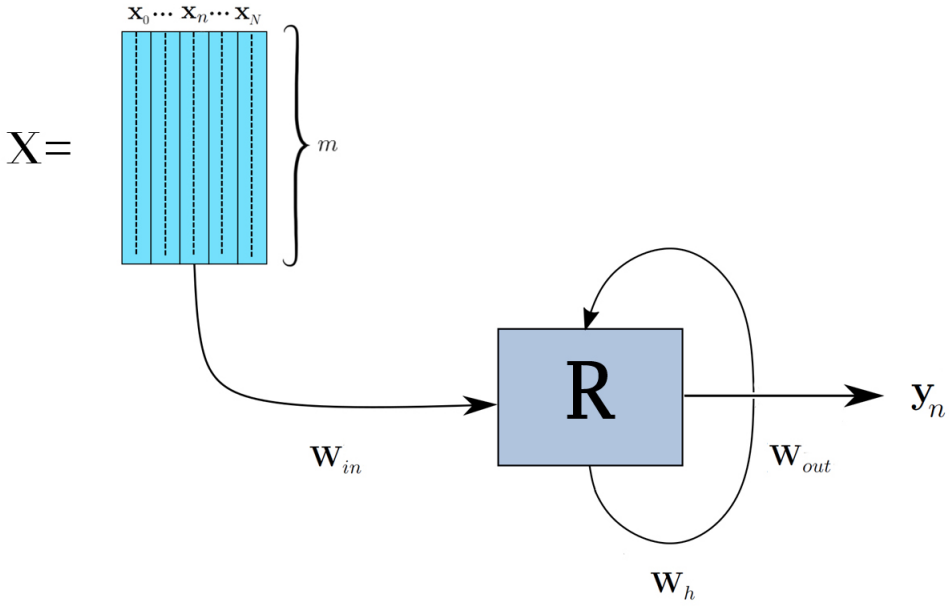

Figure 2.6 *Principle of operation in a recurrent neural network.*

Figure 2.6 illustrates the principle of operation in RNNs. Encapsulated within R is a hidden layer as in a standard MLP. The input matrix X is comprised of $N+1$ columns corresponding to a sequence of the same length. The columns from 0 to N each represent a frame in a sequence. In the case of video for example, this would simply be a flattened two-dimensional image. The diagram shows the network receiving the n^{th} frame as input at time step n. The network computes two notable outputs: the hidden state $\mathbf{h}$ and the output $\mathbf{y}$, and they are given by the equations

$$\mathbf{h}_n = \phi_h(\mathbf{W}_{in}^T \mathbf{x}_n + \mathbf{W}_h^T \mathbf{h}_{n-1}), \tag{2.39}$$

$$\mathbf{y}_n = \phi_{out}(\mathbf{W}_{out}^T \mathbf{h}_n) \tag{2.40}$$

The term $\mathbf{W}$ denotes the weights matrix (including the bias terms) and the sub-

scripts h, in and out correspond to the input, hidden, and output weights respectively. The function ϕ is the activation function. An important note to be made is that the hidden weights $\mathbf{W}_h$ connect instances of the network through time as illustrated in Figure 2.7 through the unfolded representation of the RNN. The unrolled representation also illustrates how an RNN is very similar to an MLP, with a number of hidden layers equal to the temporal length of its received input sequence in discrete time steps.

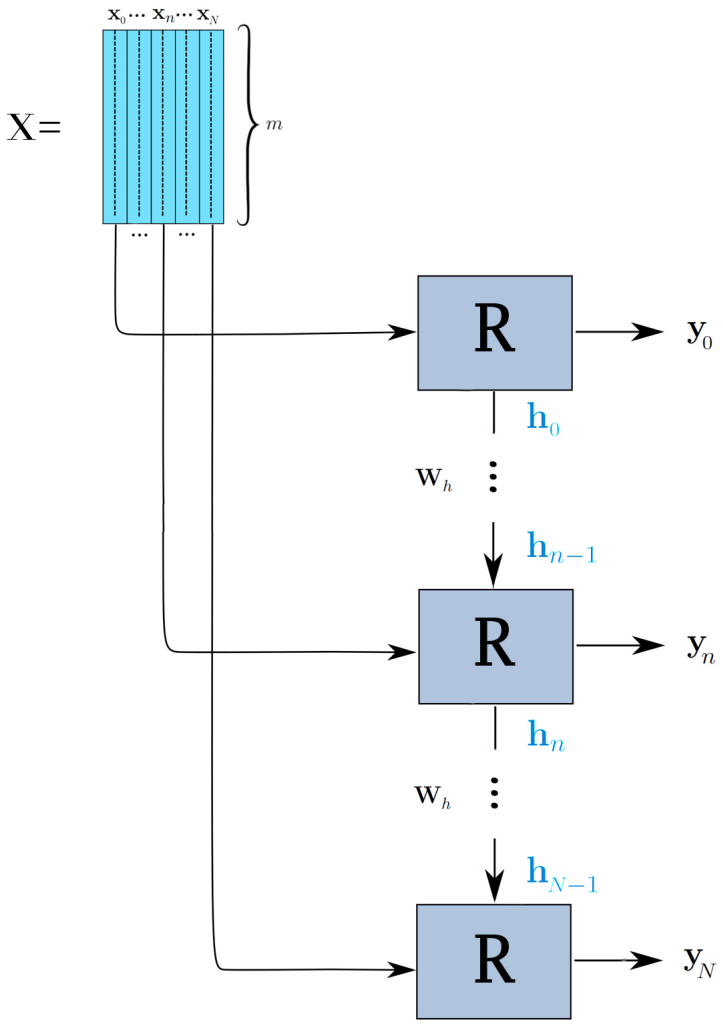

Figure 2.7 *The unrolled representation of a recurrent neural network, illustrating the flow of the hidden state through the network at future time steps.*

Note how $\mathbf{h}$ is dependent on the previous hidden state, and this dependence continues until the zeroth time step, which leads to a problem during training using a gradient-based training algorithm. Let us expand equation 2.39 for an input of temporal length three

$$\mathbf{h}_0 = \phi_h(\mathbf{W}_{in}^T \mathbf{x}_0) \tag{2.41}$$

$$\mathbf{h}_1 = \phi_h(\mathbf{W}_{in}^T \mathbf{x}_1 + \mathbf{W}_h^T \phi_h(\mathbf{W}_{in}^T \mathbf{x}_0)) \tag{2.42}$$

$$\mathbf{h}_2 = \phi_h(\mathbf{W}_{in}^T \mathbf{x}_2 + \mathbf{W}_h^T \phi_h(\mathbf{W}_{in}^T \mathbf{x}_1 + \mathbf{W}_h^T \phi_h(\mathbf{W}_{in}^T \mathbf{x}_0))) \tag{2.43}$$

We can immediately notice how this expression grows with time. This phenomenon is analyzed in more detail in the following section.

Training procedure RNNs are trained using the Back-Propagation Through Time (BPTT) algorithm [71], which is a natural extension of the BP algorithm to RNNs. The dependence of $\mathbf{h}$ on past inputs can prove problematic for BPTT. In order to illustrate this consider the expression for the derivative of the error function at time step n with respect to the weights matrix $\mathbf{W}_h$

$$\frac{\partial \mathcal{E}_n}{\partial \mathbf{W}_h} = \sum_{k=0}^{n} \frac{\partial \mathcal{E}_n}{\partial \mathbf{y}_n} \frac{\partial \mathbf{y}_n}{\partial \mathbf{h}_n} \frac{\partial \mathbf{h}_n}{\partial \mathbf{h}_k} \frac{\partial \mathbf{h}_n}{\partial \mathbf{W}_h} \tag{2.44}$$

By applying the chain rule again and taking note of the dependence of $\mathbf{h}_n$ on previous hidden states, we can rewrite equation 2.44 as $\frac{\partial \mathbf{h}_n}{\partial \mathbf{h}_k}$ as

$$\frac{\partial \mathcal{E}_n}{\partial \mathbf{W}_h} = \sum_{k=0}^{n} \frac{\partial \mathcal{E}_n}{\partial \mathbf{y}_n} \frac{\partial \mathbf{y}_n}{\partial \mathbf{h}_n} \Big(\prod_{i=k+1}^{n} \frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_{i-1}} \Big) \frac{\partial \mathbf{h}_n}{\partial \mathbf{W}_h} \tag{2.45}$$

For the purpose of this illustration, we will assume that the Jacobian enclosed within the parenthesis is diagonalizable, which then allows us to further expand it as

$$\prod_{i=k+1}^{n} \frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_{i-1}} = \prod_{i=k+1}^{n} \mathbf{W}_h^T \mathrm{diag}(\phi_h'(\mathbf{h_{i-1}})) \tag{2.46}$$

Moreover, the following result can be shown [23] about the 2-norm of the Jacobian

$$\Big\| \prod_{i=k+1}^{n} \frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_{i-1}} \Big\|_2 \leq (\eta_{\mathbf{W}} \eta_{\mathbf{h}})^{n-k} \tag{2.47}$$

The terms $\eta_{\mathbf{W}}$ and $\eta_{\mathbf{h}}$ denote the upper-bounds of $\big\| \prod_{i=k+1}^{n} \mathbf{W}_h^T \big\|_2$ and $\big\| \mathrm{diag}(\phi_h'(\mathbf{h_{i-1}})) \big\|_2$ respectively. It was shown in [23] that if these bounds are smaller than 1, their product decays exponentially at a rate proportional to the recurrence depth of the

network. Conversely, if these bounds exceed 1, then their product grows exponentially at the same rate. These two phenomena are known as the *vanishing* and *exploding* gradients respectively.

The problem of vanishing gradient has received more attention in the literature because the exploding gradient can easily be mitigated by clipping the gradients at some pre-determined threshold [23]. The vanishing gradient, on the other hand, is more difficult to handle. Several solutions have been proposed for the vanishing gradient problem such as properly initializing the weights of the network [36], using the ReLU [14] activation function, and most notably, Long Short-Term Memory networks [24].

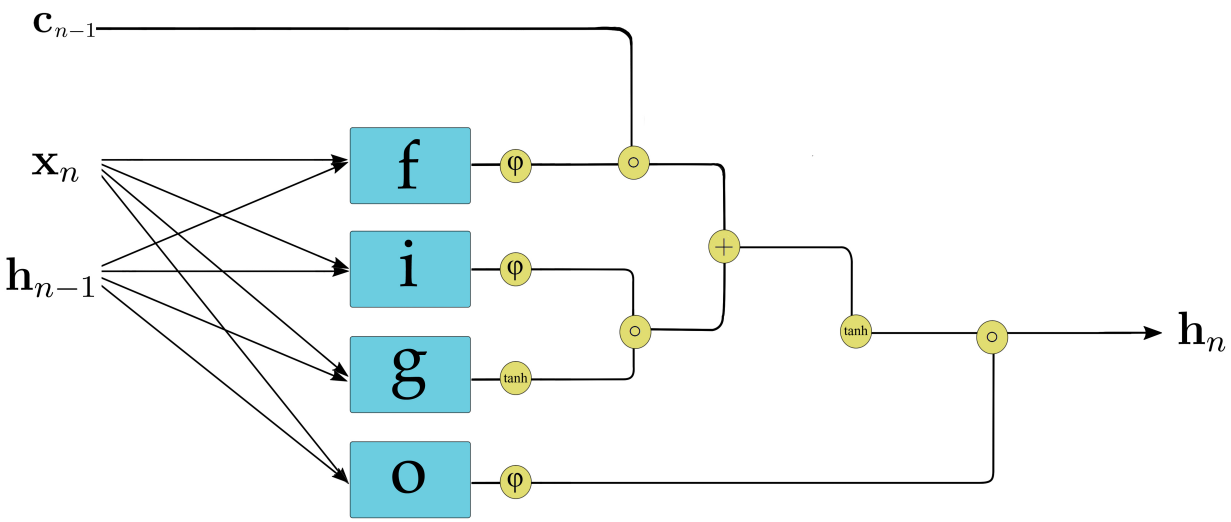

Figure 2.8 *Signal flow in an LSTM network block with the forget gate.*

LSTMs are a variant of RNNs which have been proposed for solving several problems standard RNNs suffer from, among these are learning long-term dependencies which LSTM networks are capable of for up to 1000 discrete time steps into the future [24]. More notably, LSTM networks solve the phenomena of vanishing and exploding gradients as discussed earlier. The main differences between a regular RNN and an LSTM network is the inclusion of memory states and gates. The original architecture was later extended by [12] with the inclusion of an additional *forget gate*. Figure 2.8 shows the signal flow inside an LSTM block which illustrates the computation of the following equations.

$$\mathbf{c}_n = \mathbf{i}_n \circ \mathbf{g}_n + \mathbf{f}_n \circ \mathbf{c}_n, \tag{2.48}$$

$$\mathbf{h}_n = \mathbf{o}_n \circ \tanh(\mathbf{c}_n), \tag{2.49}$$

$$\mathbf{f}_n = \phi(\mathbf{W}_f^T \mathbf{x}_n + \mathbf{U}_f^T \mathbf{h}_{n-1}), \tag{2.50}$$

$$\mathbf{i}_n = \phi(\mathbf{W}_i^T \mathbf{x}_n + \mathbf{U}_i^T \mathbf{h}_{n-1}), \tag{2.51}$$

$$\mathbf{g}_n = \phi_h(\mathbf{W}_g^T \mathbf{x}_n + \mathbf{U}_g^T \mathbf{h}_{n-1}), \tag{2.52}$$

$$\mathbf{o}_n = \phi(\mathbf{W}_o^T \mathbf{x}_n + \mathbf{U}_o^T \mathbf{h}_{n-1}), \tag{2.53}$$

where $\mathbf{i}$, $\mathbf{o}$, and $\mathbf{f}$ are known as the input, output and forget gates respectively. The element-wise multiplication operator is denoted by $\circ$. The subscript n for all variables denotes the value of that variable at time step n. The memory state in the block is denoted by $\mathbf{c}$. The weights matrices $\mathbf{W}$ and $\mathbf{U}$ are the characterizing parameters for an operation (or gate) whose respective corresponding gate is denoted by the subscript of the weights term. In essence, the main difference between LSTMs and standard RNNs is the computation of the hidden state $\mathbf{h}$. In most implementation environments such as Theano [3] and TensorFlow [1], this requires little change to the procedure of training LSTMs.

3. METHOD

This section discusses the implemented system, including data preparation, data augmentation techniques, and the model selection process.

3.1 Data Preparation

The data used in this experiment consists of about 5 hours of motion capture data in BVH format, recorded at 120 frames per second. The human model has 64 joints, including hand fingers. Several different types of motion are present in the data-set such as walking, running, lying, dancing and others. Figure 2.1 shows the distribution of motion categories in the data set. The data has not gone through post-processing, which means that recording errors are present which manifest as very fast and unnatural joint displacements. The motion of some finger joint through the z-axis is shown in Figure 3.1, with the error occurring at the highlighted segments in the second figure. The orange graph in the first figure shows the absolute-value of the second-order difference with respect to the temporal domain.

The data set has been inspected manually and motion files with severe errors have been removed and were not used in the training set. Files with motion categories that significantly deviate from most of the data set have also been removed. Examples for these include recordings of roller-skating and animations where the human model remains stationary for an extended number of frames. A portion of the training set is held for testing and validation.

3.1.1 Data Transformation

BVH files are hierarchical representations of motion as a time-series of poses. Each pose (or frame) is represented as the global translation of the root joint and the Euler rotation angles for all joints around the X, Y, and Z axes. The rotation angles are defined in the context of a pre-specified skeleton model within the file.Figure 3.3 shows an example for a human skeleton with zero rotations for all joints. The rotations specified in a motion capture data file can be mapped onto such a skeleton

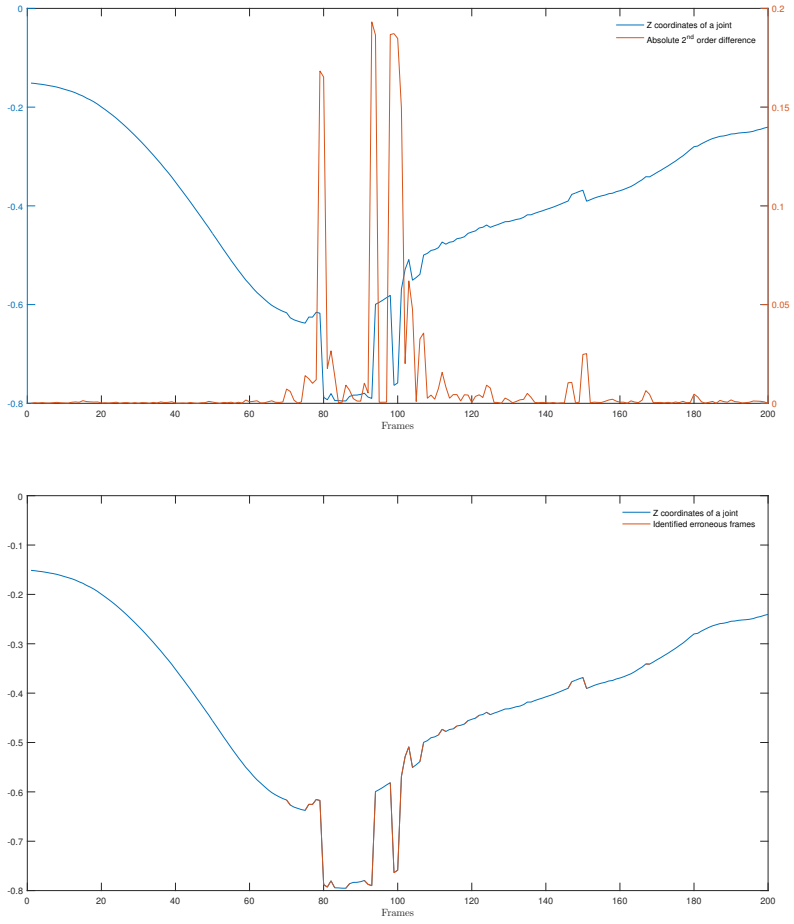

Figure 3.1 *The graphs show the z-coordinates of some joint. The orange plot highlights the sensitivity of the 2^{nd} order difference (or derivative) to the high and unnatural accelerations resulting from recording inaccuracies.*

for the purpose of animation and visualization. The skeleton can be animated by translating the entire skeleton according to the root joint translation, and then applying the appropriate joint rotations and translations as will be discussed shortly. For a thorough overview and the full BVH file format specification, the reader is referred to [40].

From experimentation, we have determined that using the joint displacements as our features produce the best results. Before we show how the joint rotations can be converted to displacements, let us first state the three rotation matrices

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sin\theta & 0 \\ 0 & sin\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{3.1}$$

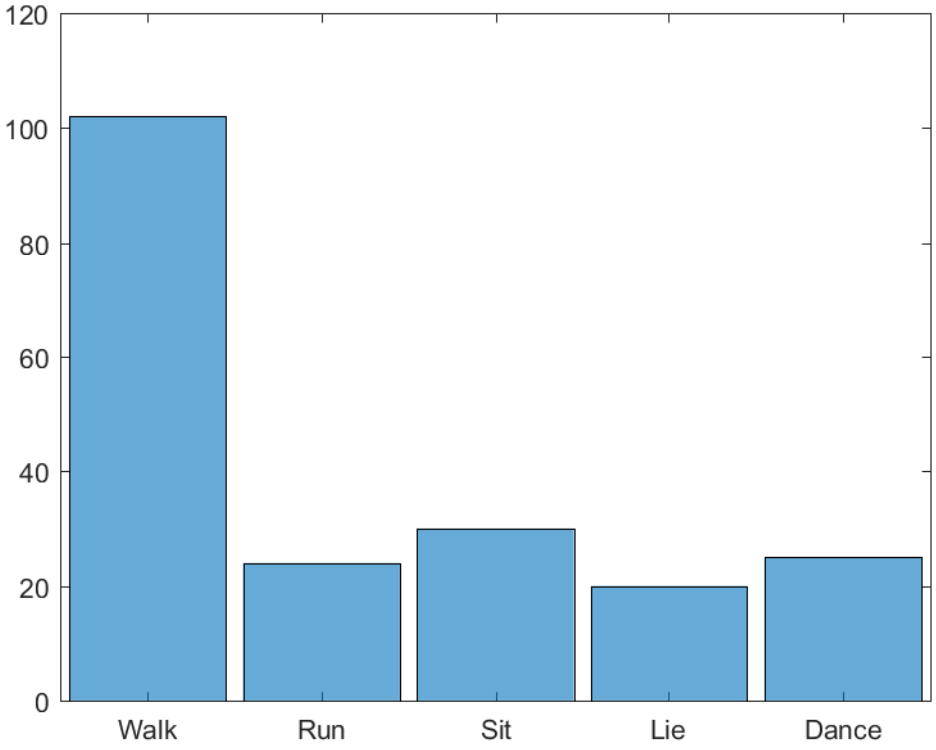

Figure 3.2 *Distribution of motion categories in the data set. It should be noted that these categories are not exclusive and that one motion file may contain one or more of these categories.*

$$R_y(\theta) = \begin{bmatrix} cos\theta & 0 & sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{3.2}$$

$$R_z(\theta) = \begin{bmatrix} cos\theta & -sin\theta & 0 & 0 \\ sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{3.3}$$

where each of R_x, R_y, and R_z are the rotation matrices that rotate a point $\mathbf{v} \in \mathbb{R}^3$ around each of the axes x, y, and z respectively. The composite rotation matrix R that rotates a point around all three axes is given by the product

$$R = R_x R_y R_z \tag{3.4}$$

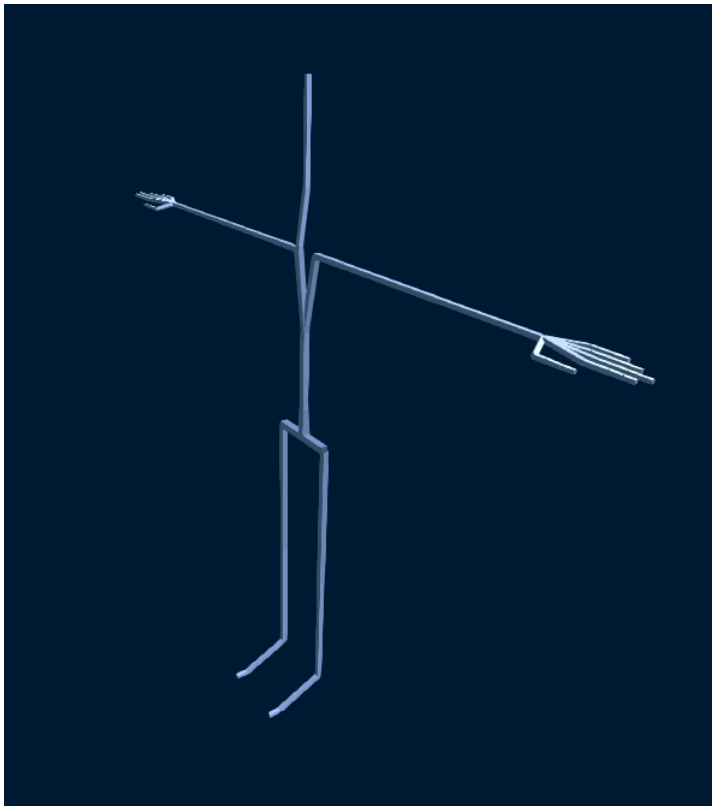

Figure 3.3 *A human skeleton model as specified in a BVH file with* 0 *rotations for all joints.*

Then the new rotated representation $\mathbf{v}'$ for the point $\mathbf{v}$ is given by

$$\mathbf{v}' = R_x R_y R_z \mathbf{v} \tag{3.5}$$

In BVH files, however, the rotations are performed in the order Z, X and Y, so our rotation transformation alternatively becomes

$$\mathbf{v}' = R_z R_x R_y \mathbf{v} \tag{3.6}$$

For translating a point $\mathbf{v}$ along the translation vector $\mathbf{t}$, the operation is stated as follows

$$\mathbf{v}' = \mathbf{t}\mathbf{v} \tag{3.7}$$

A more compact notation can be obtained by defining a transformation matrix $\mathbf{M}$, which contains the necessary information for transforming any point through

rotations and translation. The transformation matrix $\mathbf{M}$ is given as follows

$$M = \begin{bmatrix} R & R & R & t_x \\ R & R & R & t_y \\ R & R & R & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{3.8}$$

where each R is a place-holder for the elements of a rotation matrix. For example, the transformation matrix M for rotating a point by the angles ϕ, θ, ψ around the x, y, and z axes, and a linear translation by the vector $\mathbf{t}$ is given by

$$\mathbf{M} = \begin{bmatrix} cos\theta cos\psi & cos\phi sin\psi + sin\phi sin\theta cos\psi & sin\phi sin\psi - cos\phi sin\theta cos\psi & t_x \\ -cos\theta sin\psi & cos\phi cos\psi - sin\phi sin\theta sin\psi & sin\phi cos\psi + cos\phi sin\theta sin\psi & t_y \\ sin\theta & -sin\phi cos\theta & cos\phi cos\theta & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The equation for computing the position coordinates $\mathbf{s}_i$ for some joint i is given by the equation

$$\mathbf{s}'_i = \mathbf{M}_{p(i)}\mathbf{s}_i \tag{3.9}$$

$$= \mathbf{M}_{p(0)} \dots \mathbf{M}_{p(p(i))}\mathbf{M}_{p(i)}\mathbf{s}_i \tag{3.10}$$

Where $p(i)$ is a discrete-valued function that returns the parent of joint i. Each $\mathbf{M}_i$ is a *local* transformation matrix for joint i relative to its parent. Note that each transformation matrix $\mathbf{M}_i$ requires knowledge of the transformation matrix of the parent of i and so on up to the root of the hierarchy. For implementation purposes, storing each computed transformation matrix for each joint is advised in order to avoid wasting computation time. The algorithm for converting the joint rotations to displacements is shown in Algorithm 1.

For completeness, we state the scaling matrix for scaling a point by $S_x, S_y,$ and S_z along each respective axis and it is given as

Algorithm 1 Converting rotations to displacements

// These are needed to determine positions of joints in the hierarchy and their relationships
load(**offsets**);
load(**parents**);

// The argument to the function $\mathbf{r} \in \mathbb{R}^{195}$ is a vector of rotations, with the first three elements being the translation coordinates for the root

function RXYZ2DXYZ(**r**):

    d=dims(**r**); //get dimensions
    $\mathbf{t}_{root}$=**r**$[1:3,:]$;
    $\mathbf{r} = \mathbf{r}[4:end,:]$; // exclude root translations from rotation data
    f=zeros(**d**[**1**]); //stores displacements

    $n = \frac{(size(rots,1))}{3}$;

    for $i = 1 : \mathbf{d}[1]$ **do**
        M=zeros(4,4,$\frac{\mathbf{d}[1]}{3}$); // store transform mat of each joint
        $\mathbf{M}[:,:,1] = [\mathbf{R}_z(rots[3,i]) * \mathbf{R}_x(rots[1,i]) * \mathbf{R}_y(rots[2,i])\mathbf{t}_{root}[1:3,i]; 0001]$;
        $\mathbf{f}[1:3,i] = \mathbf{M}[[1,\ 2,\ 3],4,1]$;

        for $j = 2 : n$ **do**

            //$\mathbf{t}_p$ is the position coordinates of joint j's parent
            $\mathbf{t}_p = \mathbf{M}[:,:,\mathbf{parents}[j]]$;
            $\mathbf{T}_z = \mathbf{R}_z(\mathbf{r}[3 + (j-1) * 3, i])$; //**T** is a rotation matrix
            $\mathbf{T}_x = \mathbf{R}_x(\mathbf{r}[1 + (j-1) * 3, i])$;
            $\mathbf{T}_y = \mathbf{R}_y(\mathbf{r}[2 + (j-1) * 3, i])$;

            $\mathbf{M}[:,:,j] = \mathbf{t}_p * [\mathbf{T}_z * \mathbf{T}_x * \mathbf{T}_y\ \mathbf{offsets}[j,:]^T; 0\ 0\ 0\ 1]$;

            $\mathbf{f}[(1 + (j-1) * 3) : (3 * j), i] = \mathbf{M}([1\ 2\ 3], 4, j)$;

$$S = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

While no scaling operations are involved in transforming joint rotations to displacements, it is useful for visualization purposes.

Training set construction A single frame of motion is represented by a vector $\mathbf{v} \in \mathbb{R}^{192}$ vector, which is a concatenation of the x, y, and z position coordinates of each of the 64 joints. The motion capture data was thoroughly inspected and the motion contained have been categorized. Given the small size of the dataset in terms of individual files (each file may contain long sequences of motion), it was feasible to do this manually by watching and examining the content of each individual motion file at varying speeds ranging from the original 120 FPS to 200 FPS. An interactive 3D animator has been implemented to simplify the task of examining each file and categorizing its motion content.

The training set is constructed from the set of joint displacements of each file by running a sliding window of 200 frames moving at a step-size of 100 frames, resulting in a 50% overlap between consecutive windows. Figure 3.4 illustrates this process. Each input sample $\mathbf{X}$ in the training set is a 192×200 matrix, where the vertical axis contains a concatenation of all the 3-D position coordinates for each individual joint as follows.

$$\mathbf{X} = \begin{bmatrix} x_{hips}^{(1)} & x_{hips}^{(2)} & x_{hips}^{(3)} & \dots & x_{hips}^{(200)} \\ y_{hips}^{(1)} & y_{hips}^{(2)} & y_{hips}^{(3)} & \dots & y_{hips}^{(200)} \\ z_{hips}^{(1)} & z_{hips}^{(2)} & z_{hips}^{(3)} & \dots & z_{hips}^{(200)} \\ x_{spine}^{(1)} & x_{spine}^{(2)} & x_{spine}^{(3)} & \dots & x_{spine}^{(200)} \\ y_{spine}^{(1)} & y_{spine}^{(2)} & y_{spine}^{(3)} & \dots & y_{spine}^{(200)} \\ z_{spine}^{(1)} & z_{spine}^{(2)} & z_{spine}^{(3)} & \dots & z_{spine}^{(200)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{foot}^{(1)} & x_{foot}^{(2)} & x_{foot}^{(3)} & \dots & x_{foot}^{(200)} \\ y_{foot}^{(1)} & y_{foot}^{(2)} & y_{foot}^{(3)} & \dots & y_{foot}^{(200)} \\ z_{foot}^{(1)} & z_{foot}^{(2)} & z_{foot}^{(3)} & \dots & z_{foot}^{(200)} \end{bmatrix} \tag{3.11}$$

The subscript denotes the joint name according to the hierarchy defined in the BVH files. In our case, the first joint from which all other joints extend is the hips joint. The superscript denotes the frame number and it corresponds to the horizontal axis of the matrix. The joint identifiers (which are self-explanatory) for the BVH files used are listed in Table 6.1 in the APPENDIX. The identifiers denoted by the description *Endpoint* refer to the endpoint of the joint immediately preceding it, e.g., finger tips.

Each target-output sample $\mathbf{y} \in \mathbb{R}^{192 \times 1}$ in the training set is a vector, representing

a single frame of motion. This frame $\mathbf{y}$ is the frame that follows the sample $\mathbf{X} \in \mathbb{R}^{192\times200}$ extracted from the original motion file. In essence, the network is expected to complete the motion sequence it is given as input, which is a standard procedure in generative models 7. These data preparation methods result in a final data set of nearly $16,000$ samples, of which 500 have been held-out for testing and another 500 for validation.

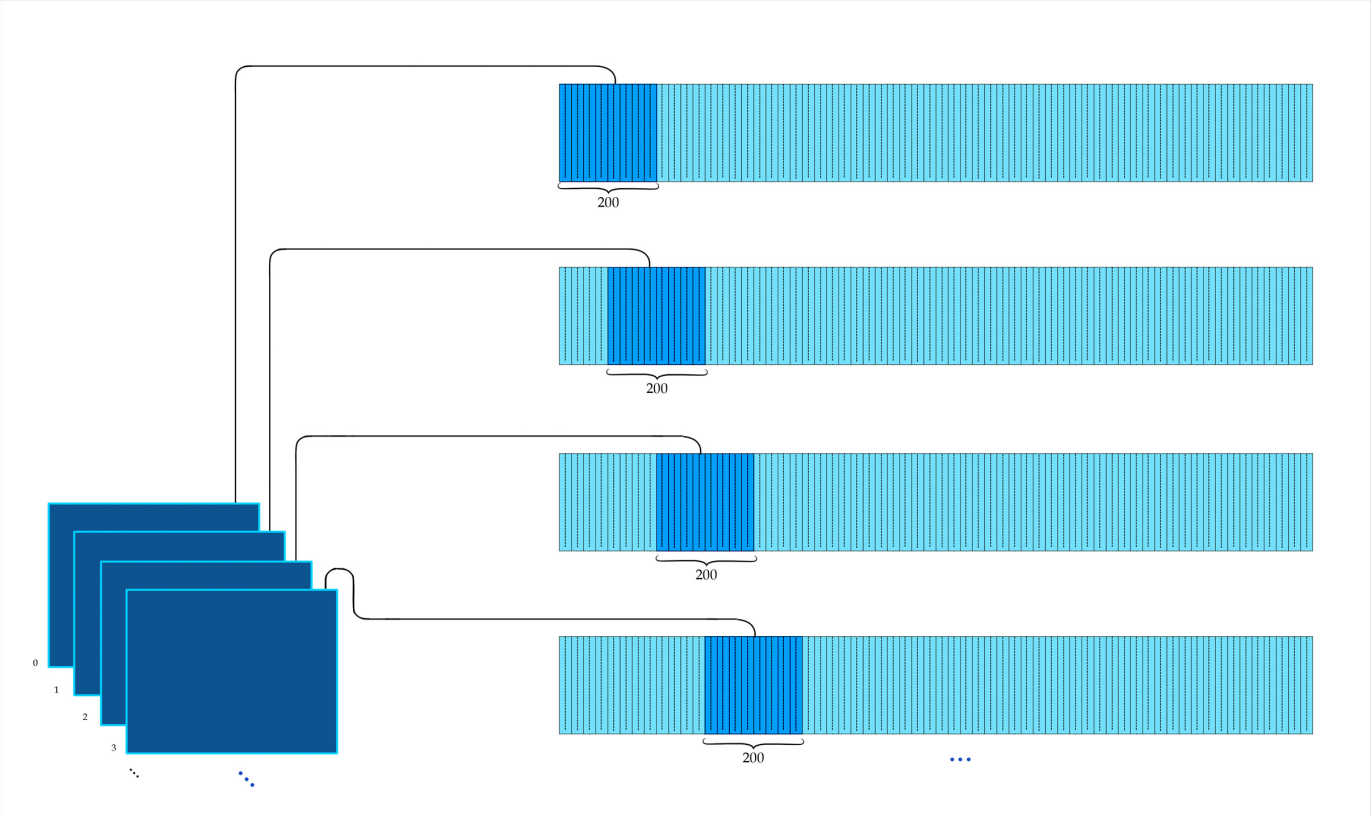

Figure 3.4 *The window of 200 frames moves at a rate of 100 frames, resulting in* 50% *overlap between two consecutive samples. Each* $\mathbf{X} \in \mathbb{R}^{192\times200}$ *is an individual training sample, shown as the dark blue squares.*

3.1.2 Proposed Data Augmentation

For motion capture data, we propose several data augmentation techniques for single-skeleton motion capture data for the purpose of motion synthesis. These data augmentation techniques preserve the qualitative attributes of the motions. Additionally, the transformations used for data augmentations are domain-preserving.

Rotation around the Y-axis The first of these techniques is rotation around the Y-axis, which preserves the qualitative features of the original motion data. Figure 3.5 shows a skeleton rotated around the Y-axis at 90-degree increments. For motion sequences with multiple skeletons, this transformation would have to be

performed for all skeleton models in the scene. This transformation can be achieved by multiplying the position coordinates for all joints over all frames by the rotation matrix $\mathbf{R}_y$. The expression for transforming a joint $\mathbf{s}_i$ by a θ-degrees rotation around the Y-axis is given as

$$\mathbf{p}_i = \mathbf{R}_y(\theta)\mathbf{p}_i \tag{3.12}$$

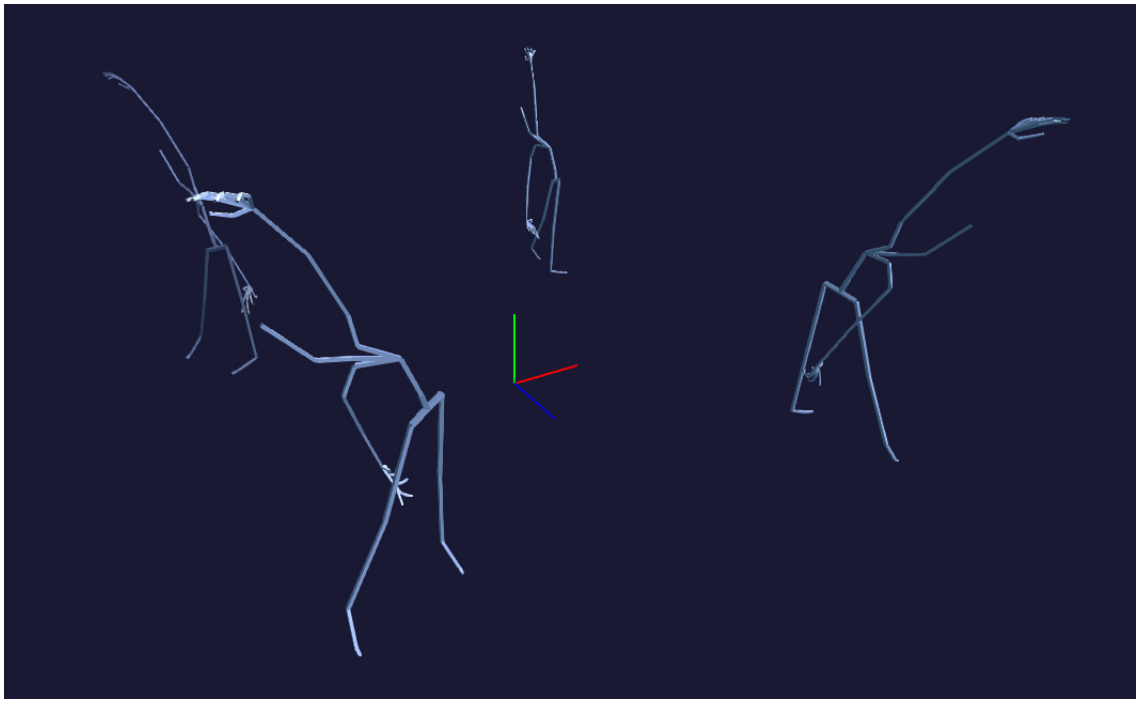

Figure 3.5 *A skeleton model rotated at increasing angles around the Y axis.*

Reflection through the XY-plane The reflection transformation is trivially achieved by flipping the sign of the z for the position coordinates pf all joints. An example for this transformation is illustrated in Figure 3.6. It should be noted that, since this transformation is a reflection, the left and right hands of a human skeleton model would be swapped. This may prove problematic for cases where left and right hand gestures are considered distinct; however, such cases can easily be remedied by reflecting the model again, and applying a global translation of the entire model to the original model's position.

Global translation Additionally, the skeleton's starting position can be changed, i.e. a linear global translation transformation over the XZ-plane. It should be noted that performing a similar transformation along the $YZ-$ and XY-planes would *not* produce valid training data, since the skeleton in these cases would either sink into the ground or float in the air, which are both unrealistic except for some rare cases such as where the human model is in mid-air. A combination of all of these proposed

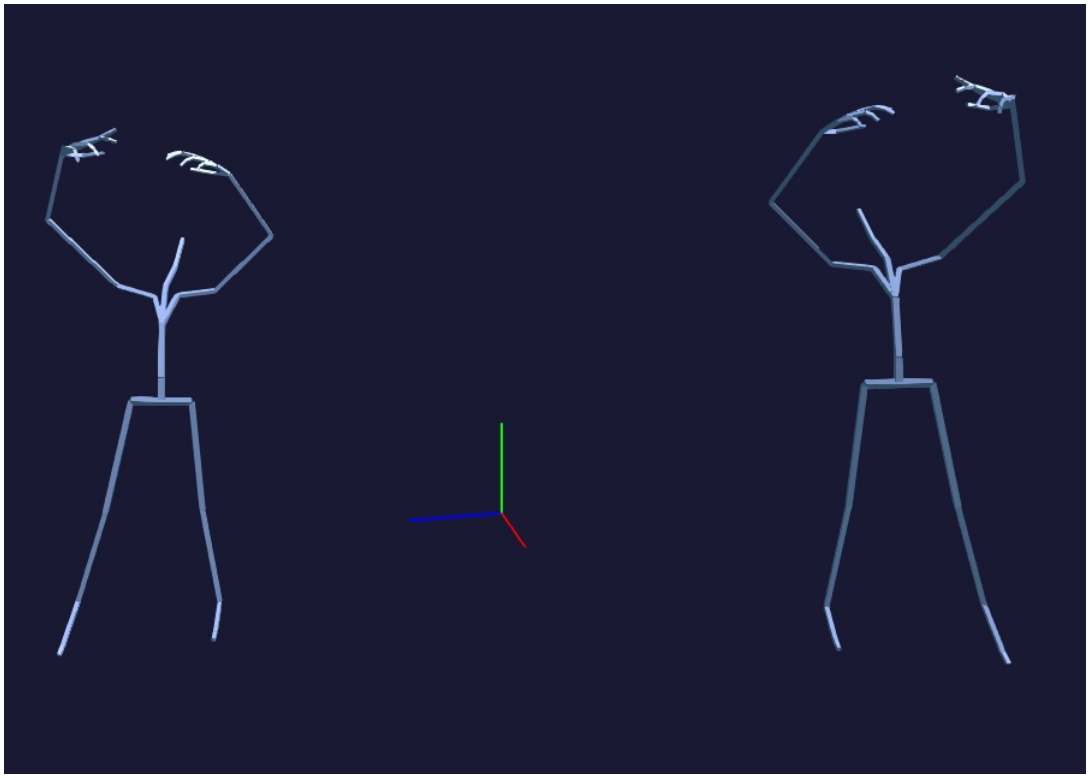

Figure 3.6 *A skeleton model reflected through the XY-plane.*

data augmentation methods may also be used, for example, combining a reflection transformation with a translation over the XZ-plane, and rotation around the Y axis.

3.2 Motion Prediction

The network accepts inputs $\mathbf{X} \in \mathbb{R}^{192} \times 200$ and outputs a single motion frame $\mathbf{y} \in \mathbb{R}^{192}$. For predicting a sequence of frames, in the base case, the input sequence $\mathbf{X}$ is simply fed to the network and the output is computed. Then, this output is concatenated with the previous input sequence $\mathbf{X}$ along the temporal axis, and it is shifted one frame forward, such that $\mathbf{X}$ will then contain the previous output $\mathbf{y}$ as its last frame while maintaining its length of 200 frames. This process is repeated for as many time steps as desired by feeding the newly constructed input sequence $\mathbf{X}$ to the network again to generate motion sequences of arbitrary length . The algorithm for this procedure is shown in Algorithm 2.

Algorithm 2 Motion Generation Procedure

```
1:
2: //S_seed ∈ R^{192×200} is an input sequence which the network uses to generate
3: //new frames on top of. The dimensionality of S_seed is dependent on the network
4: //configuration. The argument pred_N is the number of frames to be generated.
5:
6: function MOGEN(S_seed, pred_N):
7:     P=zeros(192, pred_N); //Stores sequence of predicted frames
8:     S_i = S_seed;
9:     p_i =zeros(192,1) ;
10:
11:    for i = 1 : pred_N do
12:        p_i = model.predict(S_i) // predict one frame
13:        P[:, i] = p_i; //store i^th frame
14:        S_i = [S_i[:, 1 : end]; pred_i^T)]
15:
```

3.3 Network Architecture

We train 3 LSTM networks with different architectures. We shall hereby refer to these networks by the names *LSTM1*, *LSTM2*, and *LSTM3*. The distinguishing properties of these three networks are as follows:

- *LSTM1*: the first model is a single-layered LSTM network with 1000 nodes. A dropout rate of 20% is applied to the layer.
- *LSTM2*: the second network used in the experiments has three layers of LSTM, each with 1000 neurons. A dropout rate of 20% is applied between each layer.
- *LSTM3*: the third network has three layers of LSTM just as *LSTM2*, with the exception that the first layer has 512 nodes as opposed to 1000. All other layers contain 1000 neurons. A dropout rate of 20% is used between all layers just as *LSTM2*. The purpose of this architecture was to examine the impact of encoding (reducing the dimensionality) the input on the performance of the model, and how it compares with the original *LSTM2* architecture.

The weights for all networks are initialized according to [55] by generating an orthogonal matrix with a gain factor of 1.0. The weights matrix for the recurrent kernel is initialized by sampling a truncated normal distribution; this is known as a *glorot normal initialization* [13]. The bias values are all initialized to zeros. A linear activation function is used at the output layer of the network. All hidden layers use the hyperbolic tangent as their activation functions, and a hard sigmoid

as the activation function for the recurrent step. Each network was trained until the performance of the network (in the MSE sense) stopped showing any significant improvement on the validation set. The Mean-Squared Error was used as the loss function and RMSprop [66] was chosen as an optimizer with the initial learning rate of 0.001. The mean of the loss function follows a smooth exponential decay pattern as through the epochs as illustrated in Figure 3.7. The network was trained with approximately $15,000$ samples and a mini-batch size of 32 samples.

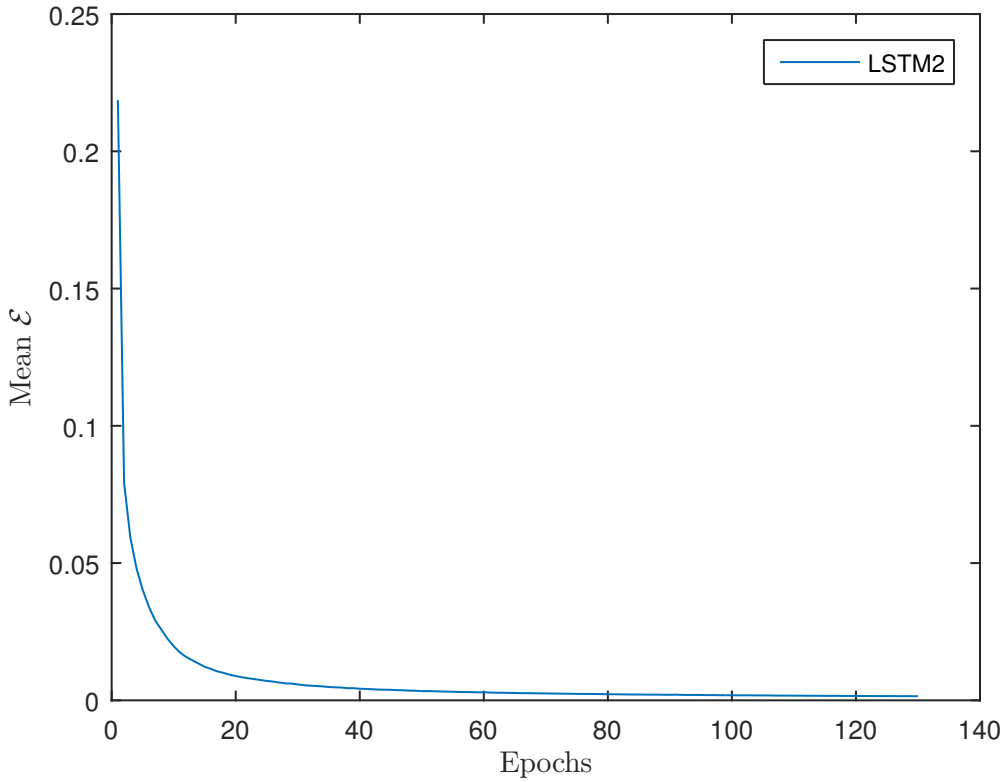

Figure 3.7 *Mean loss of the network LSTM2 which was trained for 130 epochs.*

3.4 Implementation Environment

The network was implemented and trained using the Keras neural networks library [4]. Keras runs on top of a numerical computation library such as Theano[3] or TensorFlow [1]. These numerical computation libraries provide very powerful tools for performing complex computations of mathematical expressions efficiently on the GPU or CPU. Most notably, they allow for automatic differentiation of symbolically defined expressions, which is of enormous utility for machine learning application.

The GPU used for training the network is an NVIDIA GTX 750 Ti which has a computation performance of 1305 GFLOPs. Visualizations and animations of the

skeleton have been implemented in C++ using OpenGL.

4. RESULTS AND EVALUATION

The LSTM2 network can generate novel motion sequences for a complex human skeletal model with 64 joints. Some examples of predicted frames are shown in Figure 4.1 in green. The blue frame is the last frame in the original input sequence. The skeleton in green shows the outputs of the network at varying intervals from time step 0 to 199. The predicted motion sequences show strong inter-joint and temporal joint correspondence which extends up to 250 time steps into the future.

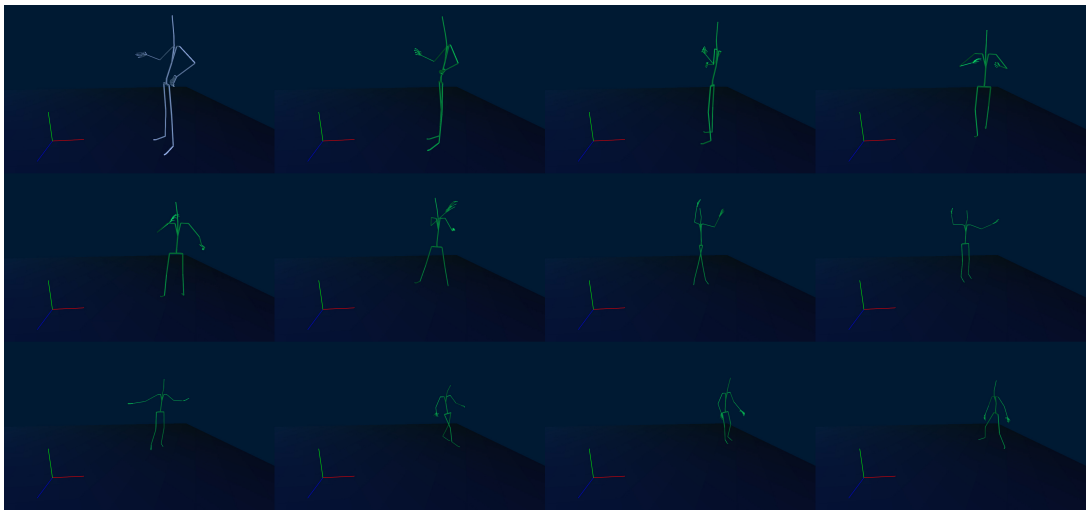

Figure 4.1 *A sample of motion frames predicted by the network shown at varying intervals for compactness and to highlight motion variety. The last frame in the sequence is the 200th predicted frame.*

Inter-joint relationships Quantifying the quality of the generated motions is difficult due to the strongly qualitative nature of human motion. We use a quantitative metric that aims to make an approximate measurement of the *correctness* of motion by evaluating the network's understanding of inter-joint relationships. This is done by computing the distance of each joint from its parent and taking the difference from the original distance of this link. Formally, for joint i, we define its Inter-Joint Correspondence Error (ICE) as

$$\mathrm{ICE}_i = \|\mathbf{s}_i - \mathbf{s}_{p(i)}\|_2 - \|\mathbf{u}_i\|_2, \tag{4.1}$$

	ICE_{avg}		
Joint	LSTM1	LSTM2	LSTM3
RightHand	0.0697	0.0459	0.0505
RightHandThumb1	0.0131	0.0134	0.0154
RightHandThumb2	0.0074	0.0061	0.0065
RightHandThumb3	0.0079	0.0062	0.006
RightHandIndex1	0.0271	0.0218	0.0249
RightHandIndex2	0.0121	0.0106	0.012
RightHandIndex3	0.0075	0.0065	0.0070
RightHandMiddle1	0.0269	0.0202	0.0230
RightHandMiddle2	0.0141	0.0112	0.0131
RightHandMiddle3	0.0078	0.0076	0.0084
RightHandRing1	0.0238	0.0173	0.0192
RightHandRing2	0.0115	0.0103	0.0121
RightHandRing3	0.0067	0.006	0.0072
RightHandPinky1	0.0217	0.0166	0.0181
RightHandPinky2	0.0101	0.008	0.0097
RightHandPinky3	0.0055	0.0053	0.0061

Table 4.1 *Table showing the ICE measurements for each model calculated for the joints in the right hand of the skeleton and averaged over all samples and all frames.*

where the vectors $\mathbf{s}_i$ and $\mathbf{s}_{p(i)}$ are the position coordinates of the joint i and the position coordinates of joint i's parent respectively. The vector $\mathbf{u} \in \mathbb{R}^3$ denotes the offset of joint i from its parent, i.e., $\|\mathbf{u}_i\|_2$ is the original length of the link connecting joints i and $p(i)$ as pre-specified for each motion data file. Table 4.1 shows the ICE measurements for the joints in the right hand of the skeleton for all three models in our experiments. We can further average this result over all the joints in the skeleton and take the absolute value to obtain the Mean Absolute Error (MAE)

$$\text{MAE}_i = \frac{1}{K+1}\sum_{j=0}^{K}\text{ICE}_j = \frac{1}{K+1}\sum_{j=0}^{K}(\|\mathbf{s}_j - \mathbf{s}_{p(j)}\|_2 - \|\mathbf{u}_j\|_2), \tag{4.2}$$

where $K+1$ is the total number of joints (this is 64 in our applications). Figure 4.3 shows the measurements of the distances between each joint and its parent (the first term in ICE), and a comparison of some example of the output of the network with an example from the training set. Figure 4.2 illustrates the MAE for each frame over all samples.

Motion content Measuring the joint relationships on their own may not always be an accurate quantification of the quality of motion, since in cases where the

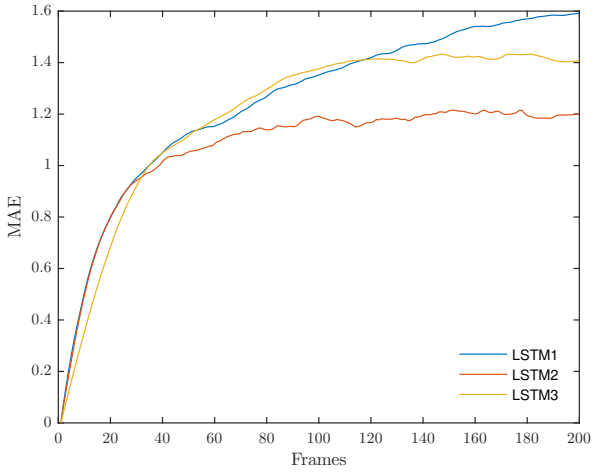

Figure 4.2 *MAE at each frame averaged over all samples and its relationship with the energy measurements.*

network outputs sequences with little to no movement, the joint-relationships-test will register it as a pass, which may not necessarily be an indicator of motion quality. Consider Figure 4.4 showing performance of each network (in the MAE sense) as we restrict the test set to a set of samples with mean energy exceeding some threshold (the horizontal axis). The purpose of this analysis is to study how the network's understanding of joint relationships changes as the energy of the motion sequence becomes larger, i.e., how well does the network maintain joint relationships as motion velocities increase. The mean energy for output sample $\mathbf{F} \in \mathbb{R}^{192\times200}$ is computed as follows:

$$E_{avg} = \frac{\sum_{j=1}^{m}\sum_{i=1}^{n} F_{i,j}}{m \times n} \tag{4.3}$$

Here, $F_{i,j}$ is the element of the first-order difference matrix $\mathbf{F} \in \mathbb{R}^{m\times n}$ at row j and column i, m is the number of degrees-of-freedom of all joints, and n is the number of frames . While equation 4.3 is a function of the velocities of the motion sequence, it does not necessarily correspond to the physical definition of energy. One can also think of this quantity as a measure of the *amount* of motion in a sequence.

The graph at the bottom in Figure 4.4 show the mean energy distribution over all samples, that is, the number of samples with a mean energy exceeding the threshold on the horizontal axis. This analysis is useful in order to verify the analysis in Figure 4.4, that is, to verify whether the mean energy distribution does not vary widely for each model, and that the analysis obtained in the top-right graph is reasonable.

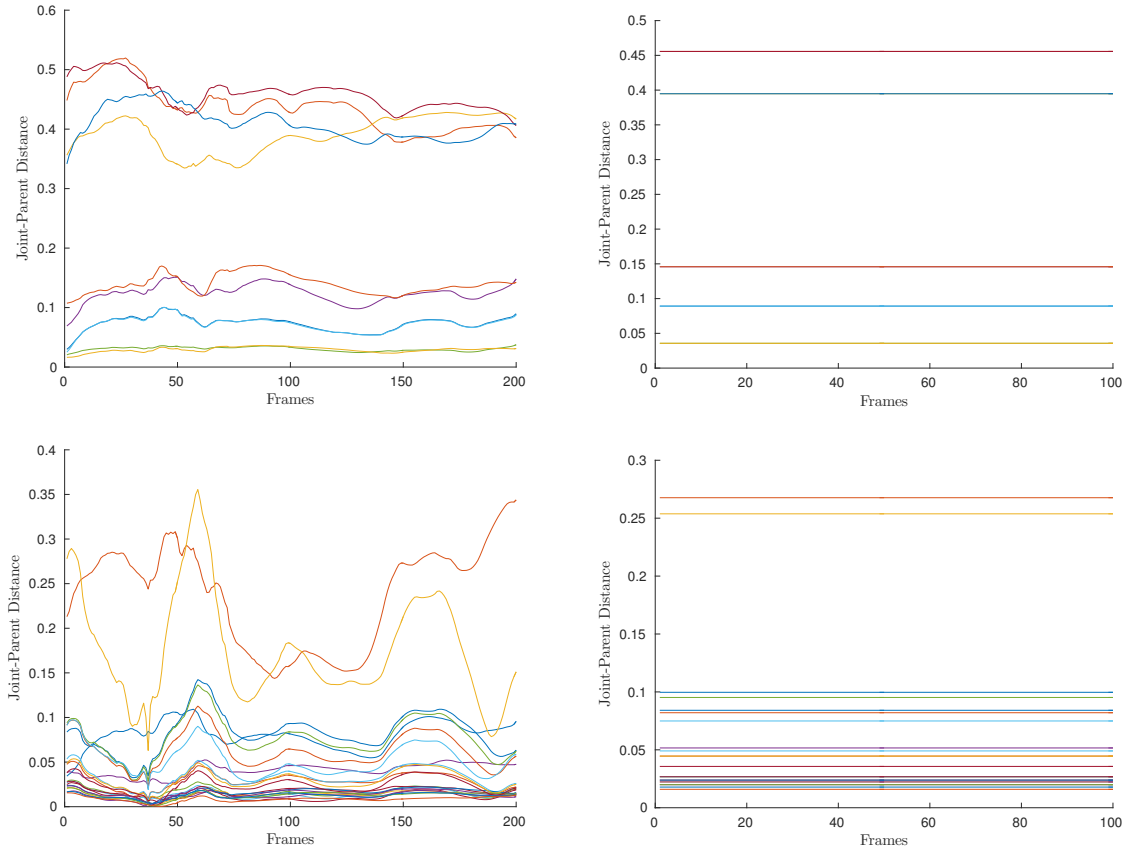

Figure 4.3 *Measurements of the distances for each joint from its parent. The pair on the top row are the joint-parent distance measurements for each joint at the lower body of the skeleton, which includes feet and legs. The pair at the bottom row shows these measurements on the right-hand joints of the skeleton. The left column are the measurements performed on output sequences generated by the network LSTM2, while the left column are for an example from the training set, which serves as the ground truth.*

Analysis and evaluation Table 4.2 shows these measurements for all samples for each model, in order to provide a general comparison of how each network understands joint relationships. MAE shows the average inter-joint correspondence error as calculated by equation 4.2. MID shows the mean inter-frame distance calculated as the vector difference between each frame and its immediately preceding frame, followed with taking the norm of each frame. The aim of the MID measurement is to evaluate the average difference (or *similarity*) between each frame. The third row, Energy_{avg} shows the average energy as calculated by equation 4.3. The last row, Energy_{std}, shows the standard deviation of the energy of all samples. It can be argued that LSTM2 shows the best capacity for learning inter-joint relationships from analyzing the MAE and the mean energy. LSTM2 produces the best MAE results while maintaining high energy variance.

Proposed applications we propose several applications for motion synthesis. Gen-

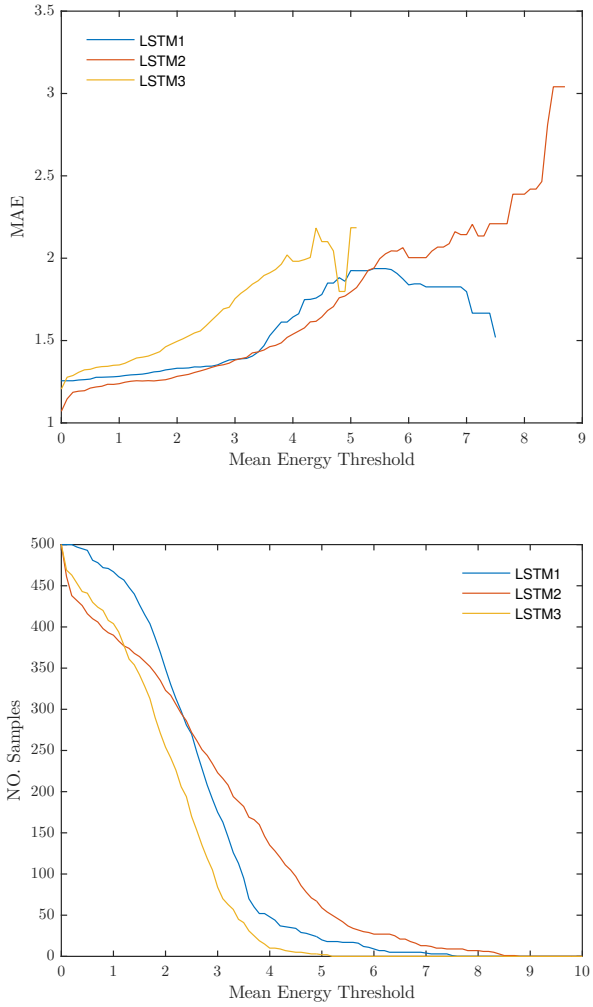

Figure 4.4 *MAE at each frame averaged over all samples and its relationship with the energy measurements.*

	Avg. Measurements Over All Samples		
Joint	LSTM1	LSTM2	LSTM3
MAE	0.0196	0.0167	0.0188
MID	0.2034	0.2185	0.1437
Energy_{avg}	2.6512	2.8149	1.9892
Energy_{std}	1.2017	1.9342	1.0882

Table 4.2 *MAE measurements averaged over all frames and all samples for each model.*

erating convincing motion sequences with some variations can be of great utility to the game industry. Scenarios where large groups of humanoids are animated can be an extremely demanding task if done manually. These techniques can also be used to produce animated scenes where the character is displaying the same animation in a repeatedly viewed scene within a video game.

In situations where new motion data is unavailable, it is possible to create new hand motion sequences where the presence of hand motion is essential, but its exact behavior and the exact gestures are not important, such as a scene where the character is playing a musical instrument, running, walking, or typing on a keyboard.

5. CONCLUSIONS AND FUTURE WORK

A machine learning technique based on recurrent neural networks has been introduced to the model 3D motion capture data for a complex human skeleton with up to 64 joints. Analysis shows that the three-layered LSTM architecture, with 1000 nodes in each layer, produces relatively optimal results when compared to the single-layer LSTM network, and the 3-layer LSTM network with a bottle-neck of 512 neurons. This network LSTM2 can generate novel motion sequences while showing good inter-joint correspondence, even for the intricate and complex hand fingers, and this correspondence extend up to 200 discrete time steps.

In the future, we aim to make use of a larger dataset and of specific motion types. Such a data set may allow the model to generate very specific motion animations that is similar to the overall motion styles in the data set. More specifically, we wish to build on the work in [7], which uses a data set of recordings of a dancer, and extend it to other motion categories such as walking, sprinting, crawling, or even different dancing styles.

Additionally, we aim incorporate to use more samples that have been transformed using the data augmentation methods proposed and to more precisely evaluate their impact on performance, motion novelty, variety and accuracy.

Generating sequences from high-level parameters has shown some promising results as demonstrated by [27] and [50]. In future endeavors, we wish to employ similar techniques to add more control on the flow and nature of the animations generated.

Using bio-mechanical constraints have shown improvements in the quality and naturalness of automatically generated animations as shown in [17]. We plan to incorporate similar bio-mechanical and physical constraints in future research.

Finally, we plan to investigate the impact of semantically-rooted regularization techniques. The intuitive motivation behind this is to allow incorrect outputs (in the MSE sense) of the network that honor inter-joint relations to have less of an impact on the direction of the gradient while training. This could however, in theory, result in a worst case scenario where the network always outputs the same pose, but

further studies are needed.

REFERENCES

[1] Allaire JJ, Eddelbuettel D, Golding N, Tang Y. tensorflow: R Interface to TensorFlow. 2016.

[2] Bayer J, Osendorfer C. Learning stochastic recurrent networks. arXiv preprint arXiv:1411.7610. 2014 Nov 27.

[3] Bergstra J, Breuleux O, Bastien F, Lamblin P, Pascanu R, Desjardins G, Turian J, Warde-Farley D, Bengio Y. Theano: A CPU and GPU math compiler in Python. InProc. 9th Python in Science Conf 2010 Jun (pp. 1-7).

[4] Chollet, F. Keras. https://github.com/ fchollet/keras, 2015.

[5] Chung J, Kastner K, Dinh L, Goel K, Courville AC, Bengio Y. A recurrent latent variable model for sequential data. In Advances in neural information processing systems 2015 (pp. 2980-2988).

[6] Ciregan D, Meier U, Schmidhuber J. Multi-column deep neural networks for image classification. InComputer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on 2012 Jun 16 (pp. 3642-3649). IEEE.

[7] Crnkovic-Friis L, Crnkovic-Friis L. Generative Choreography using Deep Learning. arXiv preprint arXiv:1605.06921. 2016 May 23.

[8] Cui X, Goel V, Kingsbury B. Data augmentation for deep neural network acoustic modeling. IEEE/ACM Transactions on Audio, Speech and Language Processing (TASLP). 2015 Sep 1;23(9):1469-77.

[9] Eck D, Schmidhuber J. A first look at music composition using lstm recurrent neural networks. Istituto Dalle Molle Di Studi Sull Intelligenza Artificiale. 2002 Mar 15;103.

[10] Elman JL. Finding structure in time. Cognitive science. 1990 Mar 1;14(2):179-211.

[11] Fragkiadaki K, Levine S, Felsen P, Malik J. Recurrent network models for human dynamics. In Proceedings of the IEEE International Conference on Computer Vision 2015 (pp. 4346-4354).

[12] Gers FA, Schmidhuber J, Cummins F. Learning to forget: Continual prediction with LSTM. Neural computation. 2000 Oct;12(10):2451-71.

[13] Glorot X, Bengio Y. Understanding the difficulty of training deep feedforward neural networks. InAistats 2010 May (Vol. 9, pp. 249-256).

[14] Glorot X, Bordes A, Bengio Y. Deep Sparse Rectifier Neural Networks. InAistats 2011 Apr 11 (Vol. 15, No. 106, p. 275).

[15] Graves A. Generating sequences with recurrent neural networks. arXiv preprint arXiv:1308.0850. 2013 Aug 4.

[16] Graves A, Jaitly N. Towards End-To-End Speech Recognition with Recurrent Neural Networks. InICML 2014 (Vol. 14, pp. 1764-1772).

[17] Geijtenbeek T, van de Panne M, van der Stappen AF. Flexible muscle-based locomotion for bipedal creatures. ACM Transactions on Graphics (TOG). 2013 Nov 1;32(6):206.

[18] Gregor K, Danihelka I, Graves A, Rezende DJ, Wierstra D. DRAW: A recurrent neural network for image generation. arXiv preprint arXiv:1502.04623. 2015 Feb 16.

[19] Grochow K, Martin SL, Hertzmann A, Popovi Z. Style-based inverse kinematics. InACM transactions on graphics (TOG) 2004 Aug 8 (Vol. 23, No. 3, pp. 522-531). ACM.

[20] Guerra-Filho G. Optical Motion Capture: Theory and Implementation. RITA. 2005;12(2):61-90.

[21] Hammer B. On the approximation capability of recurrent neural networks. Neurocomputing. 2000 Mar 31;31(1):107-23.

[22] Haykin SS. Neural networks and learning machines. Upper Saddle River, NJ, USA:: Pearson; 2009.

[23] Hochreiter S, Bengio Y, Frasconi P, Schmidhuber J. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies.

[24] Hochreiter S, Schmidhuber J. Long short-term memory. Neural computation. 1997 Nov 15;9(8):1735-80.

[25] Hochreiter S. The vanishing gradient problem during learning recurrent neural nets and problem solutions. International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems. 1998 Apr;6(02):107-16.

[26] Holden D, Saito J, Komura T, Joyce T. Learning motion manifolds with convolutional autoencoders. InSIGGRAPH Asia 2015 Technical Briefs 2015 Nov 2 (p. 18). ACM.

[27] Holden D, Saito J, Komura T. A deep learning framework for character motion synthesis and editing. ACM Transactions on Graphics (TOG). 2016 Jul 11;35(4):138.

[28] Holmboe, D., The Motion Capture Pipeline. 2008

[29] Igarashi T, Moscovich T, Hughes JF. Spatial keyframing for performance-driven animation. InProceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation 2005 Jul 29 (pp. 107-115). ACM.

[30] Jaitly N, Hinton GE. Vocal tract length perturbation (VTLP) improves speech recognition. InProc. ICML Workshop on Deep Learning for Audio, Speech and Language 2013 Jun.

[31] Kanda N, Takeda R, Obuchi Y. Elastic spectral distortion for low resource speech recognition with deep neural networks. InAutomatic Speech Recognition and Understanding (ASRU), 2013 IEEE Workshop on 2013 Dec 8 (pp. 309-314). IEEE.

[32] Krizhevsky A, Sutskever I, Hinton GE. Imagenet classification with deep convolutional neural networks. InAdvances in neural information processing systems 2012 (pp. 1097-1105).

[33] Kirk AG, O'Brien JF, Forsyth DA. Skeletal parameter estimation from optical motion capture data. InComputer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on 2005 Jun 20 (Vol. 2, pp. 782-788). IEEE.

[34] Lawrence S, Giles CL, Tsoi AC, Back AD. Face recognition: A convolutional neural-network approach. IEEE transactions on neural networks. 1997 Jan;8(1):98-113.

[35] LeCun Y, Bengio Y, Hinton G. Deep learning. Nature. 2015 May 28;521(7553):436-44.

[36] Le QV, Jaitly N, Hinton GE. A simple way to initialize recurrent networks of rectified linear units. arXiv preprint arXiv:1504.00941. 2015 Apr 3.

[37] Matsuoka K. Noise injection into inputs in back-propagation learning. IEEE Transactions on Systems, Man, and Cybernetics. 1992 May;22(3):436-40.

[38] McCulloch WS, Pitts W. A logical calculus of the ideas immanent in nervous activity. The bulletin of mathematical biophysics. 1943 Dec 1;5(4):115-33.

[39] McFee B, Humphrey EJ, Bello JP. A Software Framework for Musical Data Augmentation. InISMIR 2015 Oct (pp. 248-254).

[40] Meredith M, Maddock S. Motion capture file formats explained. Department of Computer Science, University of Sheffield. 2001;211:241-4.

[41] Montana DJ, Davis L. Training Feedforward Neural Networks Using Genetic Algorithms. InIJCAI 1989 Aug 20 (Vol. 89, pp. 762-767).

[42] Mukai T, Kuriyama S. Geostatistical motion interpolation. InACM Transactions on Graphics (TOG) 2005 Jul 31 (Vol. 24, No. 3, pp. 1062-1070). ACM.

[43] Nayebi A, Vitelli M. GRUV: Algorithmic Music Generation using Recurrent Neural Networks.

[44] Nesterov Y. A method of solving a convex programming problem with convergence rate O (1/k2). In Soviet Mathematics Doklady 1983 Feb (Vol. 27, No. 2, pp. 372-376).

[45] Ng A. CS229 Lecture notes. CS229 Lecture notes. 2000;1(1):1-3.

[46] Parkhi OM, Vedaldi A, Zisserman A. Deep Face Recognition. InBMVC 2015 Sep (Vol. 1, No. 3, p. 6).

[47] Park SI, Shin HJ, Kim TH, Shin SY. Online motion blending for realtime locomotion generation. Computer Animation and Virtual Worlds. 2004 Jul 1;15(34):125-38.

[48] Pascanu R, Mikolov T, Bengio Y. Understanding the exploding gradient problem. CoRR, abs/1211.5063. 2012 Nov.

[49] Pascanu R, Mikolov T, Bengio Y. On the difficulty of training recurrent neural networks. ICML (3). 2013 Jun 16;28:1310-8.

[50] Peng X, Berseth G, Yin K, van de Panne M. DeepLoco: Dynamic Locomotion Skills Using Hierarchical Deep Reinforcement Learning. ACM Transactions on Graphics. 2017; 36(4):41.

[51] Polyak BT. Some methods of speeding up the convergence of iteration methods. USSR Computational Mathematics and Mathematical Physics. 1964 Jan 1;4(5):1-7.

[52] Reil T, Husbands P. Evolution of central pattern generators for bipedal walking in a real-time physics environment. IEEE Transactions on Evolutionary Computation. 2002 Apr;6(2):159-68.

[53] Rose C, Cohen MF, Bodenheimer B. Verbs and adverbs: Multidimensional motion interpolation. IEEE Computer Graphics and Applications. 1998 Sep;18(5):32-40.

Rose III CF, Sloan PP, Cohen MF. ArtistDirected InverseKinematics Using Radial Basis Function Interpolation. InComputer Graphics Forum 2001 Sep 1 (Vol. 20, No. 3, pp. 239-250). Blackwell Publishers Ltd.

[54] Rosenblatt F. The perceptron: A probabilistic model for information storage and organization in the brain. Psychological review. 1958 Nov;65(6):386.

[55] Saxe AM, McClelland JL, Ganguli S. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. arXiv preprint arXiv:1312.6120. 2013 Dec 20.

Schmidhuber, J., 1992. Learning complex, extended sequences using the principle of history compression. Neural Computation, 4(2), pp.234-242.

[56] Schmidhuber J. Deep learning in neural networks: An overview. Neural networks. 2015 Jan 31;61:85-117.

[57] Schlter J, Grill T. Exploring Data Augmentation for Improved Singing Voice Detection with Neural Networks. In ISMIR 2015 (pp. 121-126).

[58] Siegelmann HT, Sontag ED. On the computational power of neural nets. Journal of computer and system sciences. 1995 Feb 1;50(1):132-50.

[59] Silaghi MC, Plnkers R, Boulic R, Fua P, Thalmann D. Local and global skeleton fitting techniques for optical motion capture. InModelling and Motion Capture Techniques for Virtual Environments 1998 (pp. 26-40). Springer Berlin Heidelberg.

[60] Simard PY, Steinkraus D, Platt JC. Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis. InICDAR 2003 Aug 3 (Vol. 3, pp. 958-962).

[61] Sims K. Evolving virtual creatures. In Proceedings of the 21st annual conference on Computer graphics and interactive techniques 1994 Jul 24 (pp. 15-22). ACM.

[62] Song HF, Yang GR, Wang XJ. Reward-based training of recurrent neural networks for cognitive and value-based tasks. eLife. 2017 Jan 13;6:e21492.

[63] Sutskever I, Martens J, Hinton GE. Generating text with recurrent neural networks. InProceedings of the 28th International Conference on Machine Learning (ICML-11) 2011 (pp. 1017-1024).

[64] Taylor GW, Hinton GE. Factored conditional restricted Boltzmann machines for modeling motion style. InProceedings of the 26th annual international conference on machine learning 2009 Jun 14 (pp. 1025-1032). ACM.

[65] Taylor GW, Hinton GE, Roweis ST. Two distributed-state models for generating high-dimensional time series. Journal of Machine Learning Research. 2011;12(Mar):1025-68.

[66] Tieleman T, Hinton G. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural networks for machine learning. 2012 Oct;4(2).

[67] Waibel A. Modular construction of time-delay neural networks for speech recognition. Neural computation. 1989;1(1):39-46.

[68] Wang JM, Fleet DJ, Hertzmann A. Gaussian process dynamical models. InNIPS 2005 Dec 5 (Vol. 18, p. 3).

[69] Webb AR. Statistical pattern recognition. John Wiley & Sons; 2003 Jul 25.

[70] Wei X, Chai J. Videomocap: modeling physically realistic human motion from monocular video sequences. InACM Transactions on Graphics (TOG) 2010 Jul 26 (Vol. 29, No. 4, p. 42). ACM.

[71] Werbos PJ. Backpropagation through time: what it does and how to do it. Proceedings of the IEEE. 1990 Oct;78(10):1550-60.

[72] Zaremba W, Sutskever I, Vinyals O. Recurrent neural network regularization. arXiv preprint arXiv:1409.2329. 2014 Sep 8.

6. APPENDIX: JOINT ID TABLE

ID	Joint	ID	Joint	ID	Joint
0	Hips	22	LeftHandRing1	44	RightHandMiddle3
1	Spine	23	LeftHandRing2	45	Endpoint
2	Spine1	24	LeftHandRing3	46	RightHandRing1
3	Neck	25	Endpoint	47	RightHandRing2
4	Head	26	LeftHandPinky1	48	RightHandRing3
5	Endpoint	27	LeftHandPinky2	49	Endpoint
6	LeftShoulder	28	LeftHandPinky3	50	RightHandPinky1
7	LeftArm	29	Endpoint	51	RightHandPinky2
8	LeftForeArm	30	RightShoulder	52	RightHandPinky3
9	LeftHand	31	RightArm	53	Endpoint
10	LeftHandThumb1	32	RightForeArm	54	LeftUpLeg
11	LeftHandThumb2	33	RightHand	55	LeftLeg
12	LeftHandThumb3	34	RightHandThumb1	56	LeftFoot
13	Endpoint	35	RightHandThumb2	57	LeftToeBase
14	LeftHandIndex1	36	RightHandThumb3	58	Endpoint
15	LeftHandIndex2	37	Endpoint	59	RightUpLeg
16	LeftHandIndex3	38	RightHandIndex1	60	RightLeg
17	Endpoint	39	RightHandIndex2	61	RightFoot
18	LeftHandMiddle1	40	RightHandIndex3	62	RightToeBase
19	LeftHandMiddle2	41	Endpoint	63	Endpoint
20	LeftHandMiddle3	42	RightHandMiddle1		
21	Endpoint	43	RightHandMiddle2		

Table 6.1 *List of all joint labels for the skeleton in the BVH files of the data set.*